Banco de Dados e Arquitetura para *Big Data*

Brasília-DF.

Elaboração

Jorge Umberto Scatolin Marques

Produção

Equipe Técnica de Avaliação, Revisão Linguística e Editoração

Sumário

Apresentação

Caro aluno

A proposta editorial deste Caderno de Estudos e Pesquisa reúne elementos que se entendem necessários para o desenvolvimento do estudo com segurança e qualidade. Caracteriza-se pela atualidade, dinâmica e pertinência de seu conteúdo, bem como pela interatividade e modernidade de sua estrutura formal, adequadas à metodologia da Educação a Distância – EaD.

Pretende-se, com este material, levá-lo à reflexão e à compreensão da pluralidade dos conhecimentos a serem oferecidos, possibilitando-lhe ampliar conceitos específicos da área e atuar de forma competente e conscienciosa, como convém ao profissional que busca a formação continuada para vencer os desafios que a evolução científico-tecnológica impõe ao mundo contemporâneo.

Elaborou-se a presente publicação com a intenção de torná-la subsídio valioso, de modo a facilitar sua caminhada na trajetória a ser percorrida tanto na vida pessoal quanto na profissional. Utilize-a como instrumento para seu sucesso na carreira.

Conselho Editorial

Organização do Caderno de Estudos e Pesquisa

Para facilitar seu estudo, os conteúdos são organizados em unidades, subdivididas em capítulos, de forma didática, objetiva e coerente. Eles serão abordados por meio de textos básicos, com questões para reflexão, entre outros recursos editoriais que visam tornar sua leitura mais agradável. Ao final, serão indicadas, também, fontes de consulta para aprofundar seus estudos com leituras e pesquisas complementares.

A seguir, apresentamos uma breve descrição dos ícones utilizados na organização dos Cadernos de Estudos e Pesquisa.

Provocação

Textos que buscam instigar o aluno a refletir sobre determinado assunto antes mesmo de iniciar sua leitura ou após algum trecho pertinente para o autor conteudista.

Para refletir

Questões inseridas no decorrer do estudo a fim de que o aluno faça uma pausa e reflita sobre o conteúdo estudado ou temas que o ajudem em seu raciocínio. É importante que ele verifique seus conhecimentos, suas experiências e seus sentimentos. As reflexões são o ponto de partida para a construção de suas conclusões.

Sugestão de estudo complementar

Sugestões de leituras adicionais, filmes e sites para aprofundamento do estudo, discussões em fóruns ou encontros presenciais quando for o caso.

Atenção

Chamadas para alertar detalhes/tópicos importantes que contribuam para a síntese/conclusão do assunto abordado.

Saiba mais

Informações complementares para elucidar a construção das sínteses/conclusões sobre o assunto abordado.

Sintetizando

Trecho que busca resumir informações relevantes do conteúdo, facilitando o entendimento pelo aluno sobre trechos mais complexos.

Para (não) finalizar

Texto integrador, ao final do módulo, que motiva o aluno a continuar a aprendizagem ou estimula ponderações complementares sobre o módulo estudado.

Introdução

A atividade de produção de *software* indiscutivelmente não tem limites, e a cada instante surge um desafio novo para os arquitetos, engenheiros e desenvolvedores. Com a grande evolução das tecnologias e produção, é notória a facilidade e a praticidade que a humanidade ganhou em acessar todo tipo de aplicação, seja qual for a finalidade. Entretanto, toda essa evolução também traz muitos desafios, problemas e/ou oportunidades.

A produção de dispositivos entrou em um *loop* no qual *hardware* e *software* disputam os holofotes para o próximo passo. Ora é lançado um *hardware* invejável em capacidade de armazenamento ou processamento, ora surge um *software* que faz brilhar os olhos de usuários em todo mundo. E isso trouxe ao ser humano uma capacidade gigantesca de produzir informações.

Hoje, cada indivíduo tem a capacidade de fazer uma revolução pela Internet, por exemplo, com um simples deslizar de dedo na tela de seu smartphone. Dificilmente teremos um dispositivo que não tenha acesso à Internet. Cada sensor, cada máquina nas indústrias, eletrodomésticos, carros e aparelhos médicos estão tendo a necessidade de trocar informação com demais dispositivos, e essa troca gera uma imensa diversidade de dados. É aí que é feita a proposta do *Big Data*.

O termo *Big Data* vai além de uma mera tradução. O conceito de *Big Data* surgiu da necessidade de nos organizarmos diante de uma enorme massa de matéria prima, rica em possibilidades de criar novos negócios e, com isso, um novo rumo para as organizações. Quando nos vimos diante de tantos dados, dos mais variados formatos, chegando em nossas mãos, foi inevitável a criação maneiras de gerenciar o enorme volume de dados que é produzido, manter confiável a variedade dos diversos formatos que estarão armazenados para que seja fácil e rápido produzir algum valor extraído dessas informações.

Por isso, estamos no auge da produção de ferramentas que estarão aptas a gerenciar esse mundo magnífico de extração de conhecimento e *insights* de novos conceitos, além É fundamental nos prepararmos o quanto antes, pois, com a velocidade que caminha essa produção, em breve surgirão seus novos *upgrades*.

Objetivos

» Introduzir o aluno ao mundo *Big Data*.

» Avaliar ferramentas e suas formas de utilização.

» Estimular o aluno a descobrir novas utilidades às ferramentas existentes.

» Mostrar ao aluno as diversas formas de se trabalhar com *Big Data*.

CONCEITOS DE *BIG DATA*

UNIDADE I

CAPÍTULO 1
Introdução

Muito antes de utilizar o termo *Big Data*, a humanidade já tinha disponível a matéria-prima para dar início à sua exploração. Os dados eram armazenados em planilhas, fichas de papel, alguns poucos bancos de dados, fitas magnéticas etc. Até então, existiam os dados brutos, assim como uma pedra preciosa, porém sem identificação e escondida, pronta para ser minerada e lapidada.

A partir dos anos 90, a era digital começa a ganhar fôlego e muitos questionamentos começam a surgir acerca da necessidade e da demanda que a Internet estaria proporcionando. Veja essa definição do guru atual do *Big Data*, Marr (2019), em seu artigo "What is *Big Data*?":

> Os computadores, e principalmente as planilhas e bancos de dados, nos deram uma maneira de armazenar e organizar dados em larga escala, de maneira facilmente acessível. De repente, as informações estavam disponíveis com o clique de um mouse. (MARR, 2019)

Bastaram as informações estarem acessíveis através da Internet e a um clique do mouse – e, mais atualmente, um toque na tela de um celular – para criarmos demandas para sua utilização. Criamos inúmeras utilidades, que serão citadas mais adiante, quando também produzimos uma quantidade inacreditável de dados. Segundo o IDC IVIEW (2012):

- De 2005 a 2021, o universo digital crescerá de 130 exabytes para 40 trilhões de gigabytes (mais de 5.200 gigabytes para cada pessoa em 2020).
- O investimento em gastos com Infra de TI cresceu em 40% entre 2012 e 2020. Como resultado, o investimento por gigabyte (GB) durante esse mesmo período cairá de US $ 2,00 a US $ 0,20.

» A maioria das informações é criada e consumida pelos consumidores – assistindo a TV digital, interagindo com a mídia social, o envio de imagens celular com câmera e vídeos entre dispositivos e ao redor da Internet, e assim por diante.

» Quase 40% da informação no universo digital será mantida por provedores de *Cloud Computing*.

» 30% da banda de internet dos USA é da Netflix; 10 bilhões de horas; 75 milhões de usuários; 15 mil títulos.

» Mais dispositivos conectados na Internet dos seres humanos; sensores, celulares, satélites, televisores etc.

Todos esses dados, seja texto ou vídeo, consomem banda de Internet, e isso possui um custo muito alto para carregá-los. Tecnicamente falando, tudo que trafega na rede passa por várias transformações. Para que possamos fazer o download de uma música e ouvi-las, por exemplo, ela passou de arquivo codificado de áudio, arquivo texto hexadecimal e, por fim, para os bits.

O bit é a menor unidade de medida digital. Os computadores trabalham sob impulsos elétricos, positivos ou negativos, que são representados por 1 ou 0. A cada impulso elétrico, também conhecido como Hertz, damos o nome de bit (***BI****nary digi****T***). Após processar os bits, normalmente são transformados e organizados logicamente para serem armazenados, ocupando, portanto, espaços de armazenamento não temporário, como os discos.

Veja a tabela a seguir, para se ter uma ideia do tamanho dos dados armazenados.

Tabela 1. Tabela de tamanho de dados.

1 Byte	8 bits
1 kilobyte (kB ou Kbytes)	1024 bytes
1 megabyte (MB ou Mbytes)	1024 kilobytes
1 gigabyte (GB ou Gbytes)	1024 megabytes
1 terabyte (TB ou Tbytes)	1024 gigabytes
1 petabyte (PB ou Pbytes)	1024 terabytes
1 exabyte (EB ou Ebytes)	1024 petabytes
1 zettabyte (ou Zbytes)	1024 exabytes
1 yottabyte (ou Ybytes)	1024 zettabytes

Fonte: O autor.

É um fato que os dados ocupam espaços de armazenamentos O que é preciso entender agora é o que significa esse “dado”.

Dados são palavras soltas, sem qualquer análise e, na maioria das vezes, não fazem sentido algum quando estiverem fora de um contexto. Eles, por si só, não transmitem nenhuma mensagem ou definem algum entendimento sobre uma situação qualquer. Por exemplo: 1/4/1980, "Alto", "São Paulo", R$ 100.000,00, uma nota musical em um arquivo de música, um pixel de uma foto etc.

Nos estudos relacionados à mineração de dados, o dado é considerado a matéria-prima, a pedra bruta a ser lapidada e transformada em uma joia valiosa. Esse processo também acontece com o dado que, quando "lapidado", tem algo de valor extraído. Esse valor que conseguimos extrair dos dados é chamado de **informação**.

Ao contrário dos dados, a informação possui significados que podem agregar valor. Segundo Davenport (2005), para que os dados possam retornar algo útil, de valor e que mostre algum sentido, ele precisar ser:

- **Contextualizado**: sabemos com que finalidade os dados foram coletados.
- **Categorizados**: conhecemos as unidades de análise ou os principais componentes de os dados.
- **Calculado**: os dados podem ter sido analisados matematicamente ou estatisticamente.
- **Corrigido**: erros foram removidos dos dados.
- **Condensado**: os dados podem ter sido resumidos de uma maneira mais forma concisa.

Assim como a informação é produzida a partir de dados dotados de algum propósito, existe um passo à frente que pode ser dado, munido de uma carga de informação a ser trabalhada e, daí, passamos a produzir o **conhecimento**. Segundo Beal (2005, p. 12), "o conhecimento também tem como origem a informação, quando a ela são agregados outros elementos". Ao citar Davenport e Prusak (2005, p. 12), Beal conceitua como uma mistura fluida de experiência condensada valores, informação contextual e *insight* experimentado, que proporciona uma estrutura para a avaliação e incorporação de novas experiências e informações.

Com as experiências relacionadas ao conhecimento, começaram a surgir as primeiras necessidades de entrarmos no processo do funcionamento do cérebro humano: o aprendizado.

O cérebro humano é formado por estruturas físicas e químicas que nos dão a capacidade de reter e transformar informações. Esse processo envolve experiências, sensações, decisões, vontades etc. e, ao final de um período que pode variar de pessoa para pessoa, podemos aprender algo que nos fará tomar decisões futuras e melhorar nosso mecanismo de aprendizado. Para captar todas essas experiências e sensações, o ser humano é dotado de sensores específicos para cada função, que levarão ao cérebro informações a serem processadas.

Alinhando o conceito de aprendizado do cérebro humano ao mundo digital, temos o que podemos chamar de Inteligência Artificial, que segue o mesmo referencial do método de aprendizado do cérebro humano, porém com suas peculiaridades. Em um resumo sobre IA, pode-se citar inúmeras formas de se obter informações e armazená-las em locais em que possam ser resgatadas, trabalhadas e/ou descartadas. Portanto, a Inteligência Artificial mais a capacidade e meios que o ser humano possui para gerar dados terão os ingredientes perfeitos para qualquer estudo sobre *Big Data*.

Outro componente ao qual foi necessário dar muita atenção foi banda de Internet. No início da popularização da Internet, as páginas eram compostas de texto puro. Existiam poucas imagens, vídeos ou áudio. Tudo era estático!

Após o ano 2000, muitas tecnologias novas surgiram e muitas foram melhoradas. Uma delas foi a própria estrutura de criação de páginas da Web. O trio HTML, CSS e Javascript passaram por mudanças e foram os grandes responsáveis por dinamizar a geração de conteúdo e informações diversas. Surgiram também consórcios entre empresas de desenvolvimento de *software* para poderem padronizar o futuro dessas tecnologias.

Ao finalizar o conceito básico de *Big Data*, não é possível passar ao próximo capítulo sem falar na estrutura básico do *Big Data*: volume, velocidade, veracidade e variedade dos dados. Alguns autores mencionam o valor como parte dessa estrutura

» Volume

Refere à quantidade absurda de dados gerados a cada segundo, a partir de vários meios, dentre eles, redes sociais, smartphones, sensores, casas

inteligentes, carros etc. A manutenção desse volume ficou inviável para ser gerenciada pelas formas tradicionais, como os bancos de dados relacionais que veremos na Unidade II. As tecnologias tradicionais trazem um grande problema no armazenamento e recuperação das informações, o que as tornam lentas e financeiramente caras.

» Velocidade

Abrange várias situações, como a velocidade com que as informações são geradas, a velocidade com que são transportadas e a velocidade com que são recuperadas; a infraestrutura de rede deve estar preparada para atender, de forma veloz, a essas demandas. O meio de transporte deverá ser eficiente o bastante para trafegar as informações do emissor até o receptor de forma rápida, eficaz e confiável, fazendo com que a velocidade de transmissão e o acesso aos dados também deve permanecer instantâneo para permitir acesso em tempo real ao site, verificação de cartão de crédito e mensagens instantâneas. A tecnologia de *Big Data* nos permite agora analisar os dados enquanto estão sendo gerados, sem nunca colocá-los em bancos de dados.

» Variedade

Pode ser definida com os tipos diferentes de dados que um laboratório de dados recebe. Hoje os dados não são mais estruturados como antigamente. Atualmente, é possível extrair algo de valor dos dados que chegam no formato texto, vídeo, planilhas, que são transmitidos por diversas fontes, como redes sociais, sites etc. A nova tecnologia de *Big Data* agora permite que dados estruturados e não estruturados sejam colhidos, armazenados e usados simultaneamente.

» Veracidade

É a qualidade ou confiabilidade dos dados. Recolher cargas e cargas de dados é inútil se a qualidade ou a confiabilidade não forem precisas. Um exemplo disso está relacionado ao uso de dados de GPS. Frequentemente, o GPS se desviará do curso enquanto você percorre uma área urbana. Os sinais de satélite são perdidos quando refletem em edifícios altos ou outras estruturas. Quando isso acontece, os dados de localização devem ser fundidos com outra fonte de dados, como dados de estradas ou dados de um acelerômetro, para fornecer dados precisos.

» Valor

Refere-se ao benefício que, depois de extraídas, as informações trarão ao negócio. Possuir uma quantidade infinita de dados não é a mesma coisa que ter algo de valor. Se os dados não puderem ser transformados em algo que trará benefícios com o uso da informação resultante, é inútil. A parte mais importante do embarque em uma iniciativa de big data é entender os custos e benefícios de coletar e analisar os dados para garantir que, em última análise, os dados coletados possam ser monetizados.

Com tantas variedades, técnicas e normas a serem estabelecidas para que o trabalho de *Big Data* seja efetivamente executado, é preciso entender como se estruturar para receber essa nova carga de material bruto. Para isso, é preciso contar com *hardware* e *software* específicos.

CAPÍTULO 2
Cluster e programação paralela

Cluster é um conjunto de computadores conectados, trabalhando como um único recurso de computação que pode criar a ilusão de ser uma máquina. Seus componentes geralmente são conectados usando redes de alta velocidade, com cada nó executando sua própria instância de um sistema operacional. Na maioria das configurações, todos os nós usam o mesmo *hardware* e o mesmo sistema operacional, embora em algumas configurações um *hardware* ou sistema operacional diferente possa ser usado. Seus nós podem ser configurados para executar a mesma tarefa, controlada e produzida por *software*.

Segundo Tanenbaum (2007, p. 10), cada *cluster* consiste em um conjunto de nós de computação controlados e acessados por um único mestre e uma de funções é manipular a alocação de nós a um determinado programa paralelo, manter uma fila de *jobs* apresentados e proporcionar uma interface aos usuários do sistema.

Definindo, portanto, *cluster* é um termo amplamente usado que pode significar uma série de computadores independentes combinados em um conjunto unificado de *hardware*, *software* e rede. Mesmo quando dois ou mais computadores são utilizados juntos no intuito de resolverem um problema, isso já é considerado um *cluster*.

A técnica de organizar os computadores em *cluster* dá suporte a diversas finalidades, como suporte de serviço na Web cálculos científicos, elaboração de chave para criptomoedas etc.

Como foi detalhado na disciplina Arquitetura Orientada a Serviços, os *clusters* podem ser utilizados para duas principais funções, Alta Disponibilidade (*HA – High Availability*) ou Alta Performance (*HPC – High Performance Computing*), com a finalidade de fornecer um poder computacional maior do que aquele fornecido por um simples computador.

Alguns autores dividem os *clusters* em classes:

» Classe 1

  Os *clusters* de Classe 1 são construídos utilizando, em sua maioria, tecnologias e peças padrão de mercado ou de fácil acesso, como interfaces de Discos SCSI, SSD, serial e placas de rede Gigabit Ethernet. O que torna seu preço mais viável que o de Classe 2.

» Classe 2

O que diferencia os *clusters* de Classe 2 são os *hardwares* utilizados em sua construção. Nele são utilizados equipamentos Blade, processadores de alta performance, normalmente menos acessíveis aos entusiastas e empresas de pequeno porte.

Aplicações e benefícios

Os benefícios que os *clusters* proporcionam são inúmeros. Por exemplo, sabendo que cada nó da rede do *cluster* é independente, a falha de um deles não significa perda de serviço.

> Um único nó é um computador independente e esse nó pode ser desativado para manutenção, enquanto o restante dos clusters assume a carga desse nó individual. Normalmente, são projetados para melhorar o desempenho e a disponibilidade dos seus serviços, além de serem muito mais econômicos do que computadores individuais com velocidade ou disponibilidade comparáveis. "A tolerância a falhas é muito utilizada em servidores web ou servidores de banco de dados em Intranets. (MORIMOTO, 2003)

Outro exemplo seria o balanceamento de carga, também poderá ser utilizado em servidores web, onde sua arquitetura é composta por pelo menos três computadores, em que um é o mestre e se encarrega de distribuir as tarefas. Esse metre não precisa, necessariamente, ser um computador com grande capacidade, o que torna essa arquitetura viável economicamente.

Para processamento paralelo existe uma tecnologia interessante chamada Beowulf. Segundo Geist *et al.* (1994):

> os *clusters* Beowulf são *clusters* de desempenho escalonáveis baseados em hardware comum, em uma rede de sistema privada, com infraestrutura de *software* de código aberto (Linux). Ele consiste em um cluster de PCs comuns ou estações de trabalho dedicadas à execução de tarefas de computação de alto processamento. A visualização dos nós no *cluster* não ficam acessíveis para os usuários, ou seja, sua única função é se dedicar à execução de trabalhos de cluster, normalmente, seu painel de configuração é exposto ao mundo externo através de apenas um nó. Alguns *clusters* do Linux são criados para oferecer confiabilidade e não velocidade.

Não existe um *software*, um pacoteou uma instalação ou um sistema operacional chamado "Beowulf". No entanto, existem vários *softwares* considerados úteis para a construção de Beowulfs que incluem os pacotes PVM e MPI, o kernel Linux etc.

O princípio de funcionamento de um *cluster* Beowulf é bastante simples: o servidor mestre, definido como *front-end*, coordena o escalonamento das tarefas entre os clientes, *back-end*, auxiliado pelas bibliotecas de troca de mensagens, como a MPI11 e PVM12 instaladas nos clientes e no servidor mestre. São as bibliotecas PVM e MPI que definem a arquitetura como sendo computação paralela.

PVM – *Parallel Virtual Machine*

O PVM é um conjunto integrado de ferramentas e bibliotecas de *software* que emula uma estrutura de computação paralela, de uso geral, flexível e heterogênea em computadores interconectados de diversas arquiteturas. O objetivo do sistema PVM é permitir que esse agrupamento de computadores interconectados seja usada cooperativamente para computação simultânea ou paralela. Segundo Geist *et al.* (1994), algumas de suas funcionalidades são, basicamente:

- ***Pool* de *hosts* configurado pelo usuário:** as tarefas computacionais do aplicativo são executadas em um conjunto de máquinas que são selecionadas pelo usuário para uma determinada execução do programa PVM. Máquinas de CPU única e multiprocessadores de hardware, incluindo computadores de memória compartilhada e de memória distribuída, podem fazer parte do pool de hosts. Após devidamente configurado, o *pool* de *hosts* pode ser alterado adicionando e excluindo máquinas durante a operação. Esse recurso é muito utilizado em projetos *failover*, ou seja, tolerância a falhas.

- **Acesso translúcido ao *hardware***: os programas aplicativos podem exibir o ambiente de hardware como uma coleção sem atributos, de elementos, de processamento virtual ou podem optar por explorar os recursos de máquinas específicas no *pool* de *hosts*, posicionando determinadas tarefas computacionais nos computadores mais apropriados.

- **Computação baseada em processos:** em particular, várias tarefas podem ser executadas em um único processador, ou seja, a unidade

de paralelismo no PVM é uma tarefa um encadeamento sequencial independente de controle que alterna entre comunicação e computação. Nenhum mapeamento de processo para processador é implícito ou imposto pelo PVM.

» **Modelo explícito de transmissão de mensagens**: coleções de tarefas computacionais, cada uma executando uma parte da carga de trabalho de um aplicativo usando decomposição de dados, funcional ou híbrida, cooperam enviando e recebendo mensagens explicitamente. O tamanho da mensagem é limitado apenas pela quantidade de memória disponível.

» **Suporte à heterogeneidade:** o sistema PVM suporta a heterogeneidade em termos de máquinas, redes e aplicativos. No que diz respeito à passagem de mensagens, o PVM permite que mensagens contendo mais de um tipo de dados sejam trocadas entre máquinas com diferentes representações de dados.

» **Suporte ao multiprocessador:** o PVM usa os recursos nativos de passagem de mensagens nos multiprocessadores para aproveitar o hardware subjacente. Os fornecedores geralmente fornecem seu próprio PVM otimizado para seus sistemas, que ainda podem se comunicar com a versão pública do PVM.

O sistema PVM é composto de duas partes:

A **primeira parte** é o *daemon* executado em segundo plano, instalando e rodando em todos os computadores que compõem a máquina virtual, projetado para que qualquer usuário autorizado possa instalá-lo em uma máquina e, quando for executado, através de um *prompt*, ele cria uma máquina virtual iniciando o PVM. Um exemplo de programa *daemon* é o programa de correio que é executado em segundo plano e lida com todo o correio eletrônico de entrada e saída em um computador. Vários usuários podem configurar máquinas virtuais sobrepostas e cada usuário pode executar vários aplicativos PVM simultaneamente.

A **segunda parte** do sistema é uma biblioteca de rotinas de interface PVM, que contém um repertório funcional e completo de primitivas necessárias para a cooperação entre as tarefas de um aplicativo. Essa biblioteca contém rotinas que podem ser chamadas pelo usuário para passagem de mensagens, processos de geração, coordenação de tarefas e modificação da máquina virtual.

O modelo de computação PVM é baseado em Threads. Quando se programa em threads é possível que um aplicativo execute várias tarefas e, cada uma dessas tarefas, é responsável por uma parte da carga de trabalho computacional do aplicativo. Aplicando paralelismo, cada tarefa executa uma função diferente, por exemplo, entrada, configuração do problema, solução, saída e exibição. Esse processo costuma ser chamado de paralelismo funcional. Um método mais comum de paralelizar um aplicativo é chamado paralelismo de dados. Dependendo de suas funções, as tarefas podem ser executadas em paralelo e podem precisar sincronizar ou trocar dados.

As linguagens de programação que o PVM suporta são C, C++ e Fortran, pois seguiu a convenção de que a maioria dos aplicativos são escritos nessas linguagens, porém com uma tendência emergente na experimentação de linguagens e metodologias baseadas em objetos.

As tarefas seguem o padrão do sistema operacional, no caso, o Linux. Elas são identificadas como um TID (*task identifier*), das quais são utilizados para o reconhecimento no envio de mensagens. Por serem únicas nas máquinas virtuais do *cluster*, elas são sustentadas pelo *daemon* do nó local (*pvmd*) e não podem ser configuradas pelo usuário. O PVM contém várias rotinas que retornam um valor tid e, assim, a aplicação do usuário pode identificar outras tarefas no sistema.

Veja o código fonte e a explicação de um programa escrito em C, disponível em http://www.netlib.org/pvm3/book/node17.html.

```
main()

{

        int cc, tid, msgtag;

        char buf[100];

        printf("i'm t%x\n", pvm_mytid());

        cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);

        if (cc == 1) {

                msgtag = 1;

                pvm_recv(tid, msgtag);

                pvm_upkstr(buf);
```

```
            printf("from t%x: %s\n", tid, buf);
        }
        else
            printf("can't start hello_other\n");
        pvm_exit();
}
```

Este programa deve ser chamado manualmente através de um terminal Linux, começa imprimindo o número da tarefa (*tid*), obtido da função *pvm_mytid()*. Dispara a cópia de outro programa chamado de *hello_other* (localizado em um nó "escravo") usando a função pvm_spawn(). Se for bem-sucedido, faz com que o programa execute uma recepção de bloqueio usando *pvm_recv*. Depois de receber a mensagem, o programa imprime a mensagem enviada por sua contraparte, bem como seu ID de tarefa; o *buffer* é extraído da mensagem usando pvm_upkstr. A chamada final pvm_exit desassocia o programa do sistema PVM.

Em seguida, veja o código do programa escravo:

```
#include "pvm3.h"
main()
{
      int ptid, msgtag;
      char buf[100];
      ptid = pvm_parent();
      strcpy(buf, "hello, world from ");
      gethostname(buf + strlen(buf), 64);
      msgtag = 1;
      pvm_initsend(PvmDataDefault);
      pvm_pkstr(buf);
      pvm_send(ptid, msgtag);
      pvm_exit();
}
```

Sua primeira rotina é obter a *tid* do "mestre" chamando a função `pvm_parent()`. Em seguida, com seu *hostname* obtido, o programa transmite para o mestre utilizando a chamada de árvore `pvm_initsend()` para a transmissão do *buffer*, colocando uma *string* através da função `pvm_pkstr()`, que será transmitida de uma maneira robusta e independente da infraestrutura e arquitetura. Isso é feito pela função `pvm_send()`, que o envia ao processo de destino especificado pela *ptid*. Dessa forma, a mensagem é enviada e sua identificação alterada para uma *tag* de valor 1.

MPI – *Message Passing Interface*

A MPI (*Message Passing Interface*) é uma biblioteca composta por métodos para troca de mensagens, que será responsável pela comunicação e sincronização de processos em um *cluster* paralelo. Em relação à programação, os processos de um programa paralelo podem ser escritos em uma linguagem de programação sequencial, tal como C, C++ ou Fortran. O objetivo principal da MPI é expor uma interface que seja largamente utilizada no desenvolvimento de programas com o propósito de troca de mensagens, garantindo sua portabilidade em qualquer arquitetura, mas sem a intenção de fornecer uma infraestrutura completa de *software* para a computação distribuída. Isso a torna recomendável para o desenvolvimento de programas paralelos de alta performance, ou seja, supercomputadores, que possuem milhares de processadores paralelos em apenas uma máquina.

Em termos simples, o objetivo da Interface de Passagem de Mensagens é fornecer um padrão amplamente usado para escrever programas de passagem de mensagens. A interface tenta ser:

- Prático.
- Portátil.
- Eficiente.
- Flexível.

Hoje, o MPI é executado em praticamente qualquer plataforma de *hardware*:

- Memória distribuída.
- Memória compartilhada.
- Híbrido.

No entanto, o modelo de programação permanece claramente um modelo de memória distribuída, independentemente da arquitetura física subjacente da máquina. Todo paralelismo é explícito: o programador é responsável por identificar corretamente o paralelismo e implementar algoritmos paralelos usando construções MPI.

Razões para usar o MPI:

» **Padronização** – MPI é a única biblioteca que tem a função de troca de mensagens. É suportado em praticamente todas as plataformas HPC. Praticamente substituiu todas as bibliotecas anteriores.

» **Portabilidade** – Há pouca ou nenhuma necessidade de modificar seu código-fonte quando você move o aplicativo para uma plataforma diferente que suporta o padrão MPI.

» **Oportunidades de desempenho** – As implementações de fornecedores devem poder explorar os recursos de *hardware* nativo para otimizar o desempenho. Qualquer implementação é livre para desenvolver algoritmos otimizados.

» **Funcionalidade** – Existem mais de 430 rotinas definidas no MPI-3, que incluem a maioria daquelas no MPI-2 e MPI-1. A maioria dos programas MPI pode ser escrita usando uma dúzia ou menos de rotinas.

» **Disponibilidade** – Diversas implementações estão disponíveis, tanto de fornecedor quanto de domínio público.

História e evolução

O MPI resultou dos esforços de numerosos indivíduos e grupos que começaram em 1992.

» **Década de 1980 – início dos anos 90**: a memória distribuída e a computação paralela se desenvolvem, assim como várias ferramentas de *software* incompatíveis para a criação de tais programas. Surgiu o reconhecimento da necessidade de um padrão.

» **Abril de 1992:** os recursos básicos essenciais para uma interface de passagem de mensagens padrão foram discutidos e um grupo de trabalho foi estabelecido para continuar o processo de padronização.

- **Novembro de 1992:** a proposta preliminar do MPI (MPI1) do ORNL é apresentada. Com o tempo, compreendeu cerca de 175 indivíduos de 40 organizações, incluindo fornecedores de computadores paralelos, criadores de *software*, acadêmicos e cientistas de aplicativos.
- **Novembro de 1993:** *Supercomputing 93 conference* – rascunho do padrão MPI.
- **Maio de 1994:** versão final do MPI-1.0 lançada.
- **Atual:** o padrão MPI-4.0 está em desenvolvimento.

CAPÍTULO 3

Infraestrutura e arquitetura de sistemas distribuídos

Um projeto em que os dados derivam de diversas fontes e em que o controle de sua produção está nas mãos de usuários da Internet, por exemplo, poderá crescer assustadoramente e necessitar de um aumento rápido de recurso. Para isso, existem formas de escalonar a disponibilidade desses recursos: a escalabilidade vertical e a escalabilidade horizontal.

A **escalabilidade vertical** possui uma desvantagem muito grande em relação a sua concorrente direta e sua forma de concepção está ligada, normalmente, aos limites físicos de uma arquitetura de *hardware* única. Ou seja, quando for necessário aumentar a capacidade de armazenamento ou de processamento, será necessário adquirir peças e componentes e acoplá-los nesse hardware.

Já a **escalabilidade horizontal** está ligada diretamente à quantidade de equipamentos de *hardware* que uma estrutura lógica e distribuída tem capacidade de receber. Ou seja, não são adquiridos peças e componentes para os equipamentos existentes, e sim novos equipamentos ligados a uma matriz de processamento. Pensando nessas estruturas distribuídas, porém gerenciada como um único modelo, algumas ferramentas foram criadas para atender o rápido crescimento que um projeto de *Big Data* necessita.

Entretanto, um fator muito importante para definirmos como o projeto irá crescer é escolher bem sua arquitetura. Segundo Tanenbaum (2007, p. 20), "os sistemas distribuídos são complexas peças de *software* das quais os componentes estão espalhados por vários computadores e, para que haja um controle eficiente, é necessário fazer a distinção lógica e física do conjunto desses componentes".

Um bom parâmetro sugerido por Tanenbaum (2007, p. 21) para iniciar os estudos sobre arquiteturas de *software* distribuídos é usar seus componentes e conectores:

- Um **componente** é uma unidade modular com interfaces bem definidas; substituível; reutilizável.
- Um **conector** é um link de comunicação entre módulos e efetua a mediação entre a coordenação ou cooperação entre os componentes.

Segundo Tanenbaum (2007, p. 21), é possível classificar os quatro mais importantes estilos arquitetônicos:

» Arquitetura em camadas

A ideia básica para o estilo em camadas é simples: os componentes são organizados de uma maneira em camadas, na qual um componente na camada Lx pode chamar componentes na camada subjacente Lx -1, mas não o contrário. Esse modelo foi amplamente adotado pela comunidade de redes. Uma observação importante é que o controle geralmente flui de camada para camada: as solicitações descem na hierarquia, enquanto os resultados fluem para cima.

» Arquitetura baseada em objetos

Uma organização muito mais flexível é seguida nas arquiteturas baseadas em objetos, na qual, em essência, cada objeto corresponde ao que definimos como componente, e esses componentes são conectados através de um mecanismo de chamada de procedimento remoto (RPC).

» Arquiteturas centradas em dados

As arquiteturas centradas em dados evoluem em torno da ideia de que os processos se comunicam por meio de um repositório comum (passivo ou ativo). Pode-se argumentar que, para sistemas distribuídos, essas arquiteturas são tão importantes quanto as arquiteturas em camadas e baseadas em objetos. Os sistemas distribuídos baseados na Web são amplamente centrados nos dados: os processos se comunicam através do uso de serviços de dados compartilhados baseados na Web.

» Arquiteturas baseadas em eventos

Nas arquiteturas baseadas em eventos, os processos se comunicam essencialmente por meio da propagação de eventos, que opcionalmente também carregam dados. Para sistemas distribuídos, a propagação de eventos geralmente tem sido associada aos sistemas de publicação/assinatura.

Arquiteturas de sistemas

Outra forma de enxergar a arquitetura distribuída é analisar como eles são organizados em componentes, considerando sua localização. Segundo Tanenbaum (2007, p. 22), "a decisão sobre os componentes de *software*,

sua interação e posicionamento leva a uma instância de uma arquitetura de *software*, também chamada de arquitetura de sistema e que está dividida em quatro arquiteturas:"

» Arquiteturas centralizadas

> Não existe um consenso para a arquitetura centralizada, porém uma solução foi elaborada em uma referência de estudo baseado na arquitetura cliente-servidor. Um servidor é um processo que implementa um serviço específico, por exemplo, um serviço de sistema de arquivos ou um serviço de banco de dados. Um cliente é um processo que solicita um serviço de um servidor enviando uma solicitação e, posteriormente, aguardando a resposta do servidor. A comunicação entre um cliente e um servidor pode ser implementada por meio de um protocolo simples sem conexão quando a rede subjacente é razoavelmente confiável, como em muitas redes locais. As arquiteturas cliente-servidor com várias camadas são uma consequência direta da divisão de aplicativos em uma interface com o usuário, componentes de processamento e um nível de dados. As diferentes camadas correspondem diretamente à organização lógica dos aplicativos.

» Arquiteturas descentralizadas

> Em muitos ambientes de negócios, o processamento distribuído é equivalente a organizar um aplicativo cliente-servidor como uma arquitetura de várias camadas, ou seja, uma distribuição vertical. A característica da distribuição vertical é que ela é obtida colocando componentes logicamente diferentes em máquinas diferentes. O termo está relacionado ao conceito de fragmentação vertical conforme usado em bancos de dados relacionais distribuídos, onde significa que as tabelas são divididas em colunas e subsequentemente distribuídas em várias máquinas." (TANENBAUM, 2007 p. 22)

Comunicação RPC

Segundo a IBM, em seu manual "Communication Programming Concepts":

> a Chamada de Procedimento Remoto (RPC) é um protocolo que fornece o paradigma de comunicação de alto nível usado no sistema operacional. O RPC pressupõe a existência de um protocolo de transporte de baixo nível, como o TCP / IP ou o UDP, para transportar os dados da mensagem

> entre os programas em comunicação. O RPC implementa um sistema lógico de comunicação cliente-servidor projetado especificamente para o suporte de aplicativos de rede.

O protocolo RPC permite que os usuários trabalhem com procedimentos remotos como se os procedimentos fossem locais. As chamadas de procedimento remoto são definidas através de rotinas contidas no protocolo RPC. Cada mensagem de chamada é correspondida com uma mensagem de resposta. O protocolo RPC é um protocolo de passagem de mensagens que implementa outros protocolos não-RPC, como lote e transmissão de chamadas remotas. O protocolo RPC também suporta procedimentos de retorno de chamada e a sub-rotina de seleção no lado do servidor.

Entendendo a comunicação RPC, é possível facilitar o entendimento da arquitetura cliente-servidor, proposta por Tanenbaum (2007, p. 22). A IBM esclarece o conceito no mesmo manual "Communication Programming Concepts":

> um cliente é um computador ou processo que acessa os serviços ou recursos de outro processo ou computador na rede. Um servidor é um computador que fornece serviços e recursos e implementa serviços de rede. Cada serviço de rede é uma coleção de programas remotos. Um programa remoto implementa procedimentos remotos. Os procedimentos, seus parâmetros e resultados estão todos documentados no protocolo do programa específico.

No RPC, cada servidor fornece um conjunto de procedimentos de serviço remoto baseado em programa e é combinado um endereço *host*, um PID e os parâmetros de entrada e saída. O cliente faz uma chamada de procedimento para enviar um pacote de dados ao servidor, quando o pacote chega, o servidor chama uma rotina de despacho, executa qualquer serviço solicitado e envia uma resposta de volta ao cliente.

Em relação ao desenvolvimento de aplicativos de comunicação RPC, o desenvolvedor precisa possuir conhecimento às teorias, bibliotecas de rede, também é útil um entendimento dos mecanismos RPC geralmente ocultos pelo compilador de protocolo do comando *rpcgen*. Entretatno, o uso do comando *rpcgen* contorna a necessidade de entender os detalhes do RPC. Várias rotinas de exemplo, APIs e códigos, poderão ser acessados no manual "Communication Programming Concepts", mencionado anteriormente.

CAPÍTULO 4
Aplicações

Até agora, foi dada uma breve introdução nos principais conceitos que compõem a retaguarda das plataformas que hospedam tecnologias robustas, capazes de executar diversas aplicações de *Big Data*. Muitas aplicações estão diretamente ligadas à Ciência de Dados e Mineração de Dados.

Neste capítulo, serão mostrados vários exemplos e aplicações de utilização de *Big Data* e como algumas empresas disponibilizam essa tecnologia para o mercado.

Plataformas IoT – *Internet of Things*

A IoT surgiu em estudos feitos no Instituto de Tecnologia de Massachusetts (MIT), por Kevin Ashton, que utilizou sensores de RFID – Identificadores por Rádio Frequência e Wi-Fi. O propósito da IoT é integrar todos os dispositivos que tenham algum sistema computacional embarcado, como celulares, automóveis, televisores, relógios etc., além de integrar vários sensores, como sensor de presença de pessoas, sensores de presença de gases, monitores cardíacos, sensores sísmicos, sensores em indústrias etc. Ou seja, todas as "coisas" estarão conectadas entre si, gerando muitos dados a todo momento, podendo proporcionar um enorme estudo sobre o comportamento humano e o cruzamento de diferentes tipos de informações sobre os clientes, como compartilhamento de informações em redes sociais, dados de geolocalização, entre outros, o que permite acompanhar seu comportamento e adequar o seu negócio para melhor atendê-las.

Empresas como Google, Microsoft e IBM já disponibilizam plataforma para testar, produzir e integrar projetos de IoT.

Plataforma Google para IoT.

https://cloud.google.com/iot-core/.

Plataforma Amazon para IoT.

https://aws.amazon.com/pt/iot/.

Plataforma Microsoft para IoT.

https://azure.microsoft.com/pt-br/overview/iot/.

Plataforma IBM para IoT.

https://www.ibm.com/br-pt/internet-of-things.

Em seu artigo denominado "Top 20 Best *Big Data* Applications & Examples in Today's World", Parijat Dutta, especialista em *Big Data* e *Hadoop*, nos presenteia com várias aplicações que utilizam *Big Data*. Seguem alguns exemplos que esse autor achou mais importantes:

Setor bancário

» Identificar os novos locais das filiais onde a demanda é alta.

» Prever a quantidade de dinheiro necessária para estar presente em uma agência na época específica de cada ano.

» Detectar atividades fraudulentas e reportar ao pessoal relacionado.

» Manipular, armazenar e analisar essa enorme quantidade de dados e garantir sua segurança também para os bancos.

Aplicação no turismo

» Analisar os dados que os viajantes fornecem nas mídias sociais.

» Alguns dos dispositivos podem coletar informações de cartão de crédito ou débito para compra rápida e identificação rápida do viajante.

» Planejar com eficiência os dados dos passageiros e suas bagagens durante toda a viagem e fornecer serviços de acordo.

» Enviar ofertas e benefícios adequados para o cliente em particular.

» Ajudar a fornecer segurança usando a tecnologia *blockchain*.

Aplicação na área da saúde

» Monitorar pacientes e enviar relatórios aos médicos associados.

» Previsão de médicos necessários em horários específicos.

» Avaliar os sintomas e identificar muitas doenças nos estágios iniciais.

» Manter os registros confidenciais protegidos e armazenar grande quantidade de dados com eficiência.

» Analisar o comportamento e a condição de saúde dos pacientes.

» Prever o local onde há chance de propagação da dengue ou da malária.

Aplicação em comércio eletrônico

» Pode coletar dados e requisitos do cliente mesmo antes do início da operação oficial.

» Cria um modelo de *marketing* de alto desempenho.

» Os proprietários de comércio eletrônico podem identificar os produtos mais visualizados e as páginas que apareceram o número máximo de vezes.

» Avalia o comportamento dos clientes e sugere produtos similares.

» Os aplicativos de *Big Data* podem gerar um relatório classificado, dependendo da idade, sexo, local do visitante etc.

Aplicação no gerenciamento de acidentes

» Pode identificar os possíveis desastres avaliando a temperatura, o nível da água, a pressão do vento e outros fatores relacionados.

» Reduzir os efeitos adversos de desastres naturais.

» Os meteorologistas podem analisar os dados coletados do satélite e do radar.

» Pode identificar o nível da água e a possibilidade de inundação em qualquer área.

» Terremotos podem ser monitorados por especialistas em gerenciamento de desastres naturais.

Aplicação na segurança nacional

» Os governos coletam as informações de todos os cidadãos e esses dados são armazenados em um banco de dados para muitos propósitos.

» A ciência de dados é implementada nesses bancos de dados para extrair informações significativas juntamente com um relacionamento oculto entre conjuntos de dados.

» Pode avaliar a densidade da população em um local específico e identificar as possíveis situações ameaçadoras antes mesmo que algo ocorra.

» Os agentes de segurança podem usar esse conjunto de dados para encontrar qualquer criminoso e detectar atividades fraudulentas em qualquer área do país.

» Além disso, o pessoal relacionado pode prever a possível disseminação de qualquer vírus ou doença e tomar as medidas necessárias para prevenir.

Aplicação em agricultura

» Aplicar em todo o processo, desde a colheita até o processo de distribuição de produtos agrícolas, como arroz, trigo, legumes e assim por diante.

» Coletar dados dos últimos anos e sugerir os pesticidas que funcionam melhor sob certas condições.

» Permite que os proprietários da empresa usem a mesma terra para vários fins e os aplicativos de ciência de dados podem gerar produção ao longo do ano sem intervalo.

» Enquanto as tecnologias inteligentes coletam dados diretamente dos campos, algoritmos avançados e ciência de dados podem impulsionar habilidades fantásticas de tomada de decisão.

Aplicação na educação

» Pode armazenar, gerenciar, analisar os grandes conjuntos de dados que incluem os registros dos alunos.

» Garantir que os documentos das perguntas não vazem antes dos exames.

» Ele fornece dados influentes sobre as atividades da sala de aula e ajuda na tomada de decisões para as organizações.

» Usando câmeras de alta resolução, imagens de vídeo e processamento de imagens, pode avaliar a expressão facial do aluno e acompanhar seus movimentos.

» Motiva os alunos, identificando problemas e prestando a melhor educação possível às crianças.

Aplicação no setor governamental

» O governo pode acessar informações funcionais diárias, considerando indecentes tópicos específicos.

» Pode ajudar a identificar as áreas que precisam de atenção e analisar para melhorar a situação atual.

» Governos podem facilmente alcançar a demanda pública e agir em conformidade.

» Ajuda a monitorar as decisões tomadas pelo governo e avaliar os resultados.

» Prever qualquer ataque terrorista e tomar as medidas necessárias para evitar condições indesejadas.

BANCO DE DADOS RELACIONAL VS BANCO DE DADOS NOSQL

UNIDADE II

CAPÍTULO 1

RDBMS - Banco de dados Relacional

Um dos métodos mais utilizados nos estudos iniciais sobre Modelagem de Dados é o criado por Peter Chen em 1976, baseado na teoria de E.F. Codd, que dizia:

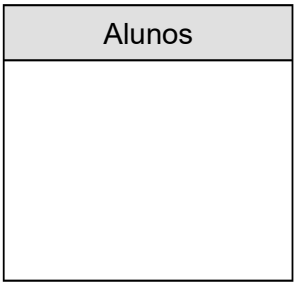
> A visão de uma dada realidade, baseia-se no relacionamento entre entidades, os quais retratam os fatos que governam esta mesma realidade, e que cada um (entidade ou relacionamento) pode possuir atributos. O conceito de abstração também aparece nesse estudo inicial e podemos citar: classificação, abstração e agregação. (CHEN, 1976)

O conceito "relacional" está caracterizado pela forma com que os componentes de uma base de dados estão relacionados. Daí o nome: base de dados relacional. Quando precisamos efetuar uma modelagem e abstrair características do mundo real para o virtual, utilizamos esses componentes, como entidades, atributos, chaves, para criarmos uma base de dados.

Esse modelo também é muito fácil de demonstrar através de esquemas ou modelos essenciais sobre o sistema ou projeto que estará sendo desenvolvido. Para deixar bem claro, é preciso salientar que no MER não existem processos, procedimentos ou fluxo de dados, ou seja, somente a informação existente no negócio deverá ser evidenciada.

As **entidades** são objetos abstraídos do mundo real e farão parte da composição do MER. Para isso, sua identificação deverá ser única e composta de seus atributos. No MER, ela será representada por um retângulo.

Figura 1. Representação de uma entidade no MER.

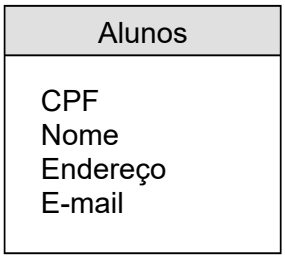


Fonte: O autor.

Os **atributos** são propriedades da entidade, ou seja, são as características dos objetos do mundo real. Eles se tornarão os campos das tabelas que armazenarão os dados na base de dados. Para compor a estrutura de atributos de uma tabela, deve-se perguntar o que descreve esse objeto no contexto do negócio a ser desenvolvido.

Por exemplo, para um sistema de escola, quais dados de alunos são necessários de armazenarmos? Os atributos podem ser nomes, valores, descrições e localizações etc.

Figura 2. Representação de atributos de uma entidade.



Fonte: O autor.

Além de armazenarem os dados brutos, determinados atributos têm funções específicas, como é o caso da **chave primária**. A chave primária ou PK (*Primary Key*) tem a função de identificar unicamente cada registro da tabela.

Existe um campo que terá a função de identificar unicamente cada informação em uma determinada tabela: chamamos de chave primária.

Na figura anterior, temos uma entidade chamada alunos e sua lista de atributos: CPF, nome, endereço, e-mail. É preciso, depois dessa lista feita, definir qual dos campos será a chave primária e, para isso, é feita a seguinte pergunta: qual desses atributos identificará unicamente um registro? Ou seja, qual conteúdo desses atributos não poderá se repetir? Na entidade alunos, poderia ser escolhido o campo CPF.

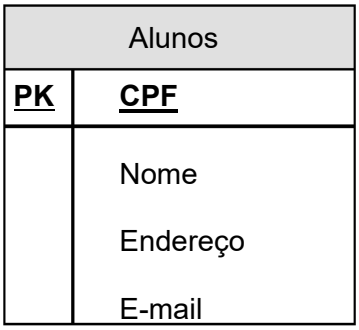
Chaves primárias são usadas para determinar que um registro não seja igual ao outro na mesma tabela e garantir que não haja redundância dos dados.

[fim atenção]

Porém, quando não exista um campo explícito para ser eleito como chave primária, é comum criar um campo para efetuar essa função. Normalmente a esse campo dá-se o nome de "código", seu tipo é número inteiro e sequencial.

Representando graficamente nosso MER, teremos o seguinte DER:

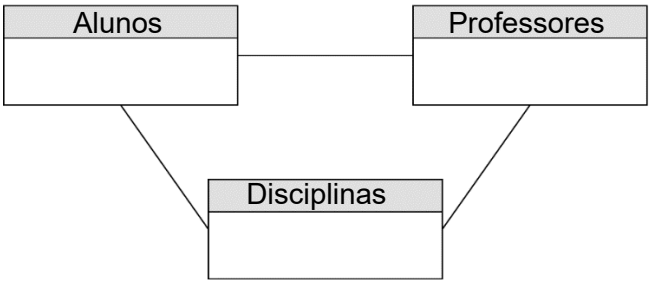
Figura 3. Entidade com chave primária.



Fonte: O autor.

Outro componente importante é justamente quem define o termo "base de dados relacional": o relacionamento. Quando é criado um MER, é importante que haja o relacionamento entre as entidades, pois certamente isso já acontece no mundo real. Por exemplo, um aluno está matriculado em uma única disciplina por período e essa disciplina é lecionada por um professor. Só nessa frase, é possível abstrair três entidades e seus relacionamentos:

Figura 4. Relacionamentos comuns.



Fonte: O autor.

Em modelagem de dados, é muito importante ter o entendimento exato do contexto do negócio para desenhar os relacionamentos entre as entidades, pois define o

acontecimento que ligam duas ou mais entidades. No exemplo anterior, o próprio pronunciamento orienta como poderá ser o relacionamento:

» Disciplina é lecionada por um único professor.

» O aluno está poderá se matricular em várias disciplinas.

» Um professor poderá lecionar em uma ou mais disciplinas.

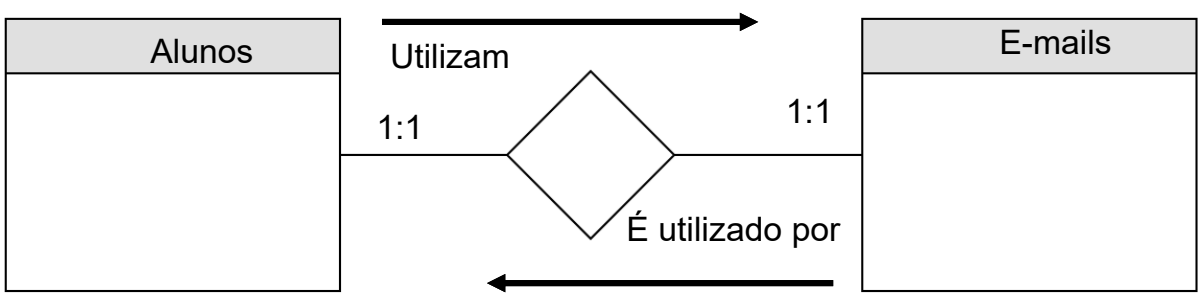
Existe um número de ocorrências nos relacionamentos entre as entidades. Esse relacionamento é feito através de campos em comum. Por exemplo: o aluno está matriculado em uma única disciplina, isso significa que um campo das entidades "alunos" e "disciplinas" se relacionarão. Quando esse fato ocorre, denominamos ***Grau do Relacionamento ou Cardinalidade,*** que influenciará no número de ocorrências entre as entidades, que, no mundo real, possuem três possiblidades:

Relacionamento um para um – 1:1

Nesse tipo de relacionamento, cada entidade só se relaciona com ***um e somente um*** elemento de outra entidade. Por exemplo: vamos supor que existe um cadastro de e-mails que serão disponibilizados aos alunos. Nesse contexto, cada aluno poderia possuir um único e-mail e o e-mail poderá ser utilizado por um único aluno. Essa ocorrência é representada por um número ao lado da entidade. Perceba também que na figura a seguir surgiu um elemento novo. O losango representa o relacionamento e sua legenda sempre será criada de acordo com o sentido da leitura.

Para representar esse relacionamento, teríamos o seguinte DER:

Figura 5. Relacionamento 1:1.



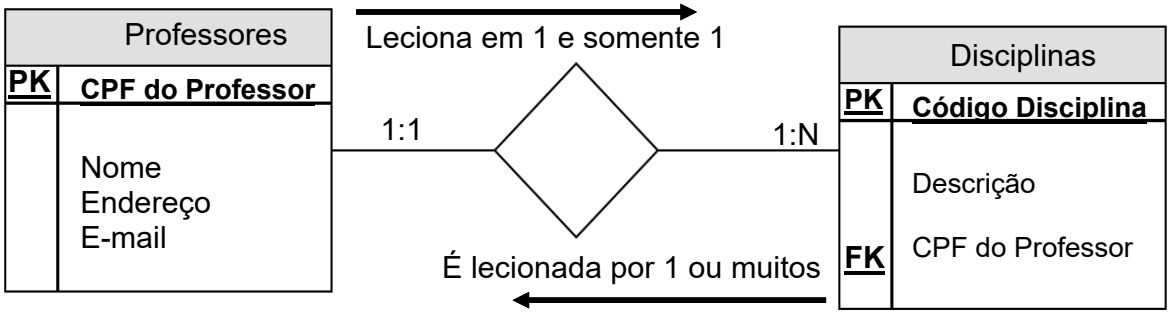
Fonte: O autor.

Relacionamento um para muitos – 1:N

É um relacionamento que acontece quando elementos de uma entidade se relacionam mais de uma vez com elementos de outra entidade. Esse é o relacionamento mais encontrado no mundo real. Por exemplo: uma disciplina é lecionada por um único professor, porém um professor poderá lecionar em várias disciplinas.

Esse tipo de relacionamento é um caso em que o sentido da leitura irá influenciar na cardinalidade e, consequentemente, o DER deverá representá-lo:

Figura 6. Sentido da leitura do relacionamento.



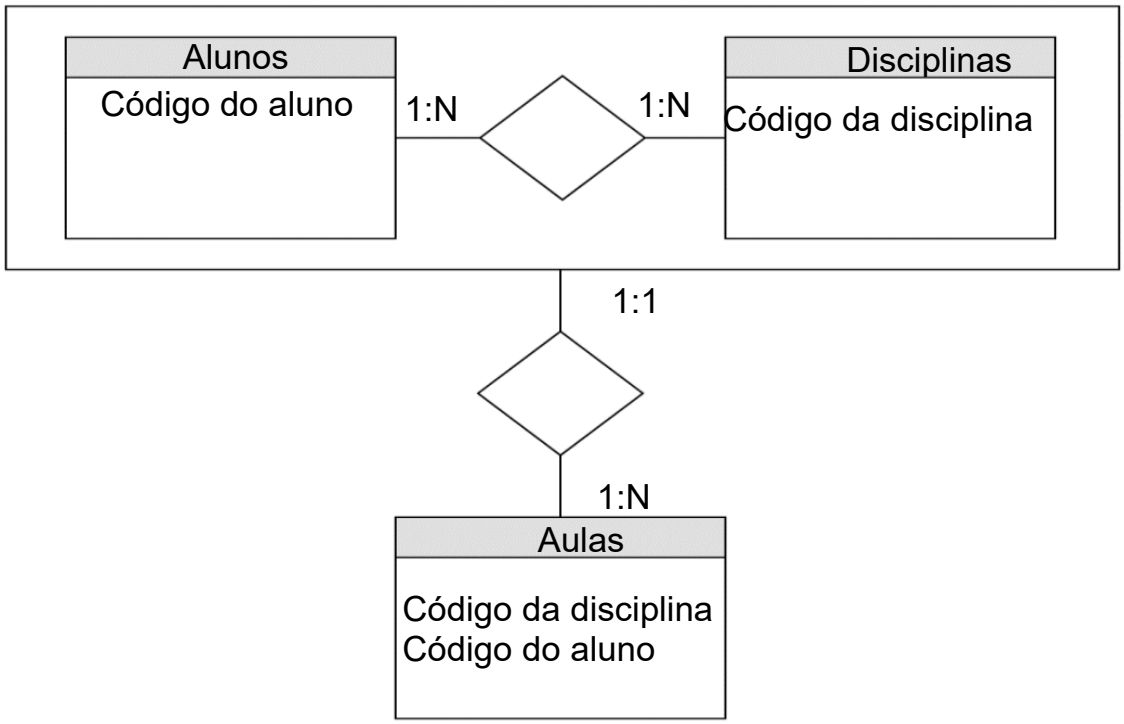
Fonte: O autor.

No relacionamento **um para muitos,** acontece um evento chamado chave estrangeira. A FK (*Foreign Key*) será o campo que, além de fazer o relacionamento entre as entidades, definirá que poderá haver várias ocorrências do campo na tabela-alvo. Chama "estrangeira" porque é um campo que não foi originalmente criado na tabela, mas migrou depois. A regra para criar uma chave estrangeira é: quando existe um relacionamento um para muitos, o campo chave primária da tabela (entidade) do lado 1 vai para o lado N como chave estrangeira.

Relacionamento muitos para muitos – N:N

O relacionamento muitos para muitos possui uma característica única, pois, seja qual for o sentido da leitura, sempre existirá um relacionamento muito para muitos. Por exemplo: um aluno participa de muitas disciplinas e em uma disciplina podem participar vários alunos. Nesse tipo de relacionamento, também há FK e suas regras: quando ocorre um relacionamento N:N, cria-se outra tabela com os campos chaves de ambas que ficará sendo a chave primária dessa tabela criada, ou seja, essa tabela possuirá uma chave primária composta com mais de um campo.

Figura 7. Resultado de relacionamento "muitos pra muitos".



Fonte: O autor.

Normalização e engenharia reversa

A normalização foi criada com o propósito de eliminar características indesejáveis, como a redundância de dados (repetição) e anomalias de inserção, atualização e exclusão. É um processo de várias etapas que coloca os dados em forma de tabela, removendo dados duplicados das tabelas de relações. A normalização é aplicada como uma abordagem sistemática de tabelas em decomposição, principalmente para:

- Eliminar dados redundantes (inúteis).
- Garantir que as dependências de dados façam sentido, ou seja, os dados são armazenados logicamente.

O criador do modelo relacional foi Edgar Codd e propôs a teoria da normalização com a introdução da Primeira Forma Normal, continuou a estender a teoria com a Segunda e a Terceira Forma Normal e, mais tarde, ele se juntou a Raymond F. Boyce para desenvolver a teoria da Forma Normal de Boyce-Codd. O processo de normalização e engenharia reversa pode ser aplicado sobre qualquer tipo de representação de dados. Ele permite obter um modelo lógico relacional a partir do modelo lógico de um banco de dados não relacional.

Permite a engenharia reversa de qualquer conjunto de dados:

» Documentos.

» Arquivos manuais.

» Arquivos convencionais em computador.

» Bancos de dados gerenciados por SGBD não relacional.

» Dados com diversos formatos.

Como base teórica para esse processo, será usado o conceito de normalização, uma técnica que objetiva eliminar redundâncias de dados de arquivos.

O primeiro passo do processo de engenharia reversa consta em transformar a descrição do documento ou arquivo a ser normalizado em um esquema de uma tabela relacional, também conhecida como:

» Tabela não-normalizada.

» Tabela aninhada.

» Grupo repetido.

» Coluna multivalorada.

» Coluna não atômica (contém tabelas aninhadas.)

Tabela 2. Dados em algum formato qualquer.

Ficha de Cadastro de Projetos					
Código: 0018	Descrição: Sistema de Estoque	Tipo: Novo Desenvolvimento	Gerente: Jonas Campos		E-mail: jonasc@escola.com
Cód. Func.	Nome Func.	Categoria	Salário	Data Início	Tempo Alocado
0001	João da Silva	A1	4	01/11/2011	20 horas
0002	Joaquim Oliveira	A1	4	01/11/2011	22 horas
0003	Maria Silveira	B1	6	01/11/2011	21 horas
0004	Pedro Duarte	A2	9	01/11/2011	23 horas
0005	Josias Santos	A2	9	01/11/2011	20 horas
0005	Marta Pereira	A1	4	01/11/2011	21 horas

Fonte: O autor.

A tabela anterior pode ser descrita pelo seguinte esquema textual de tabela relacional:

```
Projetos (CodProj, Descr, Tipo,
          (Gerente, E-mail)
          (CodFunc, NomeFunc, Cat, Sal, DataIni, TempAloc)
        )
```

O conceito de tabela aninhada é o correspondente, na abordagem relacional, aos conceitos de vetor ou "*array*" de linguagens de programação. Obtido o esquema relacional correspondente ao documento, passa-se ao processo de normalização. Este processo baseia-se no conceito de forma normal. Uma forma normal é uma regra que deve ser obedecida por uma tabela para que esta seja considerada "bem projetada". Após essa transformação, a tabela ficará na forma de tabela de dados e pronta para aplicar as formas normais, ou normalização.

Primeira Forma Normal – 1FN

Dificilmente iremos encontrar base de dados que não estejam na primeira forma normal. Entretanto, quando se inicia uma aplicação específica, na qual os dados são controlados por planilhas, é mais provável de isso acontecer.

- A primeira anomalia que se encontra nessa tabela é a anomalia de repetição de dados na inserção. Se for uma turma de 100 alunos, certamente o nome da disciplina e os dados do professor serão repetidos 100 vezes.
- Outra anomalia é a da atualização. Imagine que o professor mude seu e-mail ou a turma mude de sala. Teria que ser feita a atualização em todas as linhas. Se por acaso alguma linha deixasse de ser alterada, teria uma inconsistência nos dados.
- A anomalia de exclusão é focada em dados repetidos. Se por acaso o professor for excluído, os dados permanecerão inconsistentes.

Para que uma tabela esteja na primeira forma normal, as etapas devem ser seguidas:

1. A tabela deve ter apenas atributos com valor único (atômico).
2. Os valores armazenados em uma coluna devem ter o mesmo domínio.
3. Todas as colunas em uma tabela devem ter nomes exclusivos.
4. Quando não contém tabelas aninhadas.

É possível começar solucionando o exemplo anterior utilizando a decomposição:

1. É criada uma tabela na 1FN referente à tabela não normalizada e que contém apenas as colunas com valores atômicos, isto é, sem as tabelas aninhadas.
2. Para cada tabela aninhada, é criada uma tabela na 1FN composta pelas seguintes colunas:
   - A chave primária de cada uma das tabelas na qual a tabela em questão está aninhada.
   - As colunas da própria tabela aninhada.
3. São definidas as chaves primárias das tabelas na 1FN que correspondem a tabelas aninhadas.

Com isso, pode-se utilizar duas opções:

» Construir uma única tabela com redundância de dados.

```
Projetos (
CodProj, Descr, Tipo, Gerente, E-mail, CodFunc,
NomeFunc, Cat, Sal, DataIni, TempAloc)
```

» Construir uma tabela para cada tabela aninhada.

```
Projetos (CodProj, Tipo, Descr)
ProjetoEmpregados (CodProj, CodFunc, Nome, Cat, Sal,
DataIni, TempAl)
```

Dependência funcional

Um atributo ou conjunto de atributos A é dependente funcional de um outro atributo B contido na mesma entidade, se a cada valor de B existir nas linhas da entidade em que aparece um único valor de A. Ou seja, A depende funcionalmente de B.

Salário depende funcionalmente da coluna “código” pelo fato de cada valor de código estar associado sempre ao mesmo valor de “salário”. O valor “A1” da coluna “código” identifica sempre o mesmo valor de salário (“4”). Para denotar esta dependência funcional, usa-se uma expressão na forma:

$$\text{Código} \rightarrow \text{Salário}.$$

Segunda Forma Normal – 2FN

Objetiva eliminar um certo tipo de redundância de dados. Uma tabela encontra-se na segunda forma normal (2FN) quando, além de encontrar-se na primeira forma normal, cada coluna não chave depende da chave primária completa.

Para uma tabela estar na segunda forma normal:

» Deve estar na 1FN.

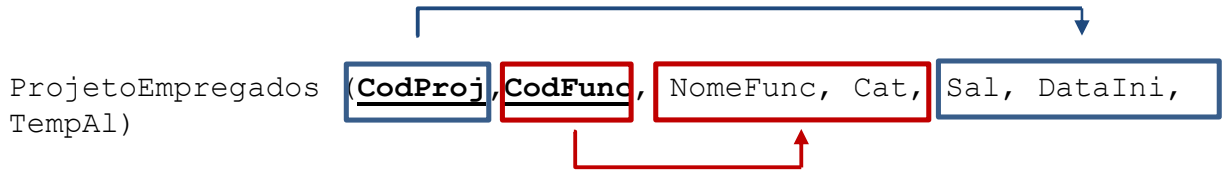
» Não deve ter dependência parcial.

A dependência parcial existe, quando, para uma chave primária composta, qualquer atributo da tabela depende apenas de uma parte da chave primária e não da chave primária completa. Para solucionar o problema da Dependência Parcial, é preciso dividir a tabela, remover o atributo que está causando dependência parcial e movê-lo para outra tabela na qual ele se encaixa bem.

Uma tabela que está na 1FN e que possui apenas uma coluna como chave primária não contém dependências parciais.

Projetos (CodProj, Tipo, Descr)

A tabela ProjetoEmpregados necessitará de uma análise, pois possui chave primária composta:



» As colunas Nome, Cat e Sal dependem, cada uma, apenas da coluna CodFunc.

» Nome, Cat e Sal são determinados tão somente pelo número do empregado.

» As colunas DataIni e TempAl dependem da chave primária completa.

Para passar à 2FN, isto é, para eliminar as dependências de parte da chave primária, é necessário dividir a tabela **ProjetoEmpregados** em duas tabelas:

ProjetoEmpregados (**CodProj,CodFunc**, DataIni, TempAl)

Empregados **(CodFunc**, NomeFunc, Cat, Sal)

Terceira Forma Normal – 3FN

Diz-se que uma tabela está na terceira forma normal quando:

» Está na 2FN.

» Não possui dependência transitiva.

Dependência transitiva é quando uma coluna não chave primária depende funcionalmente de outra coluna ou combinação de colunas não chave primária. A passagem à 3FN consta em dividir tabelas de forma a eliminar as dependências transitivas:

Categorias (**Cat**, Sal)

Ao final, teremos o seguinte modelo:

Projetos (**CodProj**, Tipo, Descr)
ProjetosEmpregados (**CodProj, CodFunc**, DataIni, TempAl)
Empregados (**CodFunc**, NomeFunc, Cat)
Categorias (**Cat**, Sal)

A normalização é um mecanismo que poderá ser utilizada em Banco de Dados NoSQL, que veremos no próximo capítulo.

CAPÍTULO 2
NoSQL

Não há dúvidas de que a tecnologia de banco de dados relacional domina o mercado e, arriscamos a falar, dominará por um bom tempo. Isso pelo fato de, sempre que iniciado um projeto de banco de dados, o processo existente é pelo relacionamento de entidades. Outro indicador é os constantes lançamentos de versões dos bancos de dados existentes no mercado, como PostgreSQL, Oracle, SQLServer, FireBird etc.

Apesar da tecnologia de bancos de dados relacionais serem dominantes no mercado, algumas tecnologias vêm surgindo para suprir as dificuldades encontradas no relacional e não para substituí-las. Uma delas é a NoSQL.

Os bancos de dados NoSQL, "Not Only SQL" (Não Apenas SQL – em português), surgiram da necessidade de demanda em que os bancos de dados relacionais são ineficazes, pois são superiores em performance e escalabilidade. Então um banco de dados NoSQL é uma alternativa ao padrão SQL que não adote o paradigma relacional de alto desempenho e ainda oferece melhor robustez e escalabilidade que o padrão SQL. Eles usam vários modelos de dados, incluindo documento, gráfico, chave-valor, memória e pesquisa. Esses tipos de banco de dados são otimizados especificamente para aplicativos que exigem modelos de grande volume de dados, baixa latência e flexibilidade.

Os bancos de dados NoSQL são ideais para muitos aplicativos modernos, como dispositivos móveis, Web e jogos, que exigem bancos de dados flexíveis, escaláveis, de alta performance e altamente funcionais para proporcionar ótimas experiências aos usuários.

Veja algumas considerações sugeridas por uma das gigantes em tecnologia de banco de dados NosSQl, a Amazon (2019):

» Flexibilidade

  Os bancos de dados NoSQL geralmente fornecem esquemas flexíveis que permitem um desenvolvimento mais rápido e iterativo. O modelo de dados flexível torna os bancos de dados NoSQL ideais para dados semiestruturados e não estruturados.

» Escalabilidade

Os bancos de dados NoSQL geralmente são projetados para serem escalados horizontalmente usando clusters distribuídos de *hardware*, em vez de escalá-los verticalmente adicionando servidores caros e robustos. Alguns provedores de nuvem lidam com essas operações nos bastidores como um serviço totalmente gerenciado.

» Alta performance

O banco de dados NoSQL é otimizado para modelos de dados específicos (como documento, chave-valor e gráfico) e padrões de acesso que permitem maior performance do que quando se tenta realizar uma funcionalidade semelhante com bancos de dados relacionais.

» Altamente funcional

Os bancos de dados NoSQL fornecem APIs e tipos de dados altamente funcionais criados especificamente para cada um de seus respectivos modelos de dados.

O próprio site do MongoDB (2019) menciona algumas vantagens:

» Grandes volumes de dados estruturados, semiestruturados e não estruturados.

» Sprints ágeis, iteração rápida e envio frequente de códigos.

» Programação orientada a objetos que é fácil de usar e flexível.

» Arquitetura eficiente e dimensionável em vez de arquitetura monolítica cara.

Nos bancos de dados relacionais, os esquemas precisam ser definidos antes, ou seja, para que os dados possam ser inseridos, é preciso que as tabelas, campos etc. sejam criados. Em um desenvolvimento ágil, isso pode ser o gargalo entre o crescimento do negócio e o time de desenvolvimento.

Isso piora quando o banco de dados é grande, pois essas alterações de estruturas podem tornar o aplicativo indisponível e atrapalhar a organização que estiver utilizando.

Os bancos de dados NoSQL são criados para permitir a inserção de dados sem um esquema predefinido, facilitando alterações significativas nos aplicativos

em tempo real, sem se preocupar com interrupções de serviço, proporcionando um desenvolvimento mais rápido, a integração de código é mais confiável e é necessário menos tempo para o administrador do banco de dados.

NoSQL *versus* RDBMS

Veja algumas características entre os bancos de dados NoSQL e bancos de dados relacionais:

Tabela 3. Comparação entre NoSQL e RDBMS.

NoSQL	RDBMS
MongoDB, Cassandra, HBase, Neo4j	MySQL, PostgreSQL, banco de dados Oracle
Desenvolvido nos anos 2000 para lidar com as limitações dos bancos de dados relacionais, particularmente em relação a escala, replicação e armazenamento de dados não estruturados.	Desenvolvido na década de 1970 para lidar com a primeira vaga de aplicativos de armazenamento de dados.
Os bancos de dados de chave-valor, documento, gráfico e em memória do NoSQL são projetados para OLTP, aplicativos de baixa latência e dados semiestruturados.	Bancos de dados relacionais são projetados para aplicativos transacionais e fortemente consistentes de processamento de transações online (OLTP) e são bons para processamento analítico online (OLAP).
Normalmente dinâmico, com algumas regras de validação de dados de aplicação. Os aplicativos podem adicionar novos campos rapidamente e, ao contrário das linhas da tabela SQL, dados diferentes podem ser armazenados juntos, conforme necessário.	O modelo relacional normaliza dados em tabelas, compostas por linhas e colunas. Um esquema define estritamente tabelas, colunas, índices, relações entre tabelas e outros elementos do banco de dados. O banco de dados impõe a integridade referencial nos relacionamentos entre as tabelas.
A performance geralmente é uma função do tamanho do cluster do hardware subjacente, da latência de rede e do aplicativo que faz a chamada.	A performance normalmente depende do subsistema do disco. A otimização de consultas, índices e estrutura de tabela é necessária para alcançar máxima performance.

Fonte: https://www.mongodb.com/nosql-explained.

Outra grande desvantagem do RDBMs para o modelo NoSQL é o controle transacional ACID:

» Atomicidade

Uma transação é garantida pela atomicidade, ou seja, ela garante que todas as tarefas sejam executadas em conjunto ou, em caso de falha, nenhuma tarefa seja executada. Se uma parte da tarefa falhar, todas as outras serão desfeitas.

» Consistência

A consistência garante a integridade dos dados através das exigências de segurança do SGDB. Ela evita redundância, falha e que sejam realmente gravados no banco de dados.

» Isolamento

O isolamento é praticado nas ações do SGBD e permite que os dados sejam gravados de maneira independente sem interferir umas nas outras e que todas aconteçam em sua ordem e por completo.

» Durabilidade

Permite que os dados gravados no banco de dados sejam duráveis, que eles estejam gravados após as operações de *Commit* e possam ser acessados por tempo indefinido.

As operações *Commit* e *Rollback* são projetadas para serem disparadas ao final das transações, nas quais o *Commit* efetiva a gravação dos dados no banco de dados, quando não houver anomalias nas transações. Caso algum problema ocorra, toda a transação é desfeita através da operação de *Rollback*.

Teorema de CAP

O teorema do CAP se aplica a sistemas distribuídos que armazenam estado. O teorema afirma que os sistemas de dados compartilhados em rede só podem garantir/apoiar fortemente dois dos três vínculos adequados a seguir:

» Consistência

Garantia de que todos os nós em um *cluster* distribuído retornem à gravação mais recente e bem-sucedida. Existem vários tipos de modelos de consistência. A consistência no CAP (usada para provar o teorema) refere-se à linearizabilidade ou consistência sequencial, uma forma muito forte de consistência.

» Disponibilidade

Cada nó que não falha retorna uma resposta para todas as solicitações de leitura e gravação em um período de tempo razoável.

» Tolerante a partições

O sistema continua funcionando e mantém suas garantias de consistência, apesar das partições de rede. Os sistemas distribuídos que garantem a tolerância da partição podem continuar funcionando normalmente depois que a partição se recuperar.

O teorema do CAP é responsável por instigar a discussão sobre as várias compensações em um sistema de dados compartilhados distribuídos. Ele desempenhou um papel fundamental no aumento de nossa compreensão dos sistemas de dados compartilhados. No entanto, o teorema da PAC é criticado por ser muito simplista e muitas vezes enganoso.

CAPÍTULO 3
Modelagem de dados NoSQL

Tipos de banco de dados NoSQL

Geralmente, os bancos de dados NoSQL incluem as seguintes famílias:

Documentos

Os chamados sistemas de banco de dados orientados a documentos são caracterizados pelo fato de sua organização de dados não necessitar de esquemas. Isso significa que:

- Os registros não precisam ter uma estrutura uniforme, ou seja, registros diferentes podem ter colunas diferentes.
- Os tipos dos valores de colunas individuais podem ser diferentes para cada registro.
- As colunas podem ter mais de um valor (matrizes).
- Os registros podem ter uma estrutura aninhada.

Os armazenamentos de documentos geralmente usam notações internas, que podem ser processadas diretamente em aplicativos, principalmente JSON. Os documentos JSON podem ser armazenados como texto puro em armazenamentos de chave-valores.

Repositórios de gráficos

São usados para armazenar informações sobre redes e grafos. Exemplo: conexões sociais. Alguns bancos de dados de mercado são: AllegroGraph, IBM Graph, Neo4j e Titan.

Armazenamento em colunas largas

Organizam as tabelas de dados como colunas em vez de linhas. Os armazenamentos de colunas largas podem ser encontrados nos bancos de dados SQL e NoSQL. Os armazenamentos de colunas amplas podem consultar grandes volumes de dados mais rapidamente que os bancos de dados relacionais convencionais.

Os armazenamentos de chave-valor

Os armazenamentos de "chave-valores" são os tipos mais básicos de bancos de dados NoSQL, em que cada chave é única e constituída de listas, conjuntos e *hashes*. Os valores podem ser tabelas *hash*, *string*, JSON e BLOB. Nele, cada item no banco de dados é armazenado como um nome ou chave de atributo, juntamente com seu valor, permitindo que o desenvolvedor armazene os dados sem esquemas. Esse tipo de armazenamento pode ser usado como coleções, dicionários, matrizes associativas etc.

Um dos grandes desafios da modelagem de dados é equilibrar as necessidades de recursos do aplicativo, as características de desempenho do SGBD e os padrões de recuperação de dados. Ao projetar modelos de dados, é fundamental considerar o uso dos dados pelo aplicativo, como consultas, atualizações e processamento dos dados e a estrutura inerente dos próprios dados.

Por padrão, os banco de dados NoSQL não exigem que seus documentos tenham um esquema, diferentemente dos bancos de dados SQL, nos quais você deve determinar e declarar o esquema de uma tabela antes de inserir dados, ou seja, os documentos em uma única coleção não precisam ter o mesmo conjunto de campos e o tipo de dados para um campo pode diferir entre os documentos dentro de uma coleção e, para alterar a estrutura dos documentos em uma coleção, como adicionar novos campos, remover campos existentes ou alterar os valores dos campos para um novo tipo, atualize os documentos para a nova estrutura. Essa flexibilidade facilita o mapeamento de documentos para uma entidade ou um objeto. Cada documento pode corresponder aos campos de dados da entidade representada, mesmo se ele tiver uma variação substancial de outros documentos na coleção. Na prática, no entanto, os documentos em uma coleção compartilham uma estrutura semelhante e você pode impor regras de validação de documentos para uma coleção durante as operações de atualização e inserção.

Os documentos incorporados capturam os relacionamentos entre os dados, armazenando dados relacionados em uma única estrutura de documento. Esses modelos de dados desnormalizados permitem que os aplicativos recuperem e manipulem dados relacionados em uma única operação de banco de dados.

As referências armazenam os relacionamentos entre os dados, incluindo links ou referências de um documento para outro. Os aplicativos podem resolver essas referências para acessar os dados relacionados. Em termos gerais, esses são modelos de dados normalizados. No exemplo a seguir, o documento "endereço" e o documento "acesso" contêm uma referência ao documento "aluno".

```
{
    _id: <ObjectId1>,
     nome:”Joao da Silva”,
     endereco:{
                rua: “Rua Duque de Caxias”,
                cidade:”São Paulo”
       },
    acesso:{
              nivel: 3,
              grupo: “alunos”
       }
}
```

Em geral, use modelos de dados incorporados quando:

» Existir relacionamentos um-para-um entre entidades.

» Existir relacionamentos um-para-muitos entre entidades. Nessas relações, os “muitos”, ou documentos filhos, sempre aparecem ou são visualizados no contexto dos “um”, ou documentos pai.

Veja no exemplo a seguir:

```
{
   _id :  “jose” ,
   nome :  “Jose da Silva”
}

{
   endereco_id :  “jose” ,
   rua :  “Duque de caxias” ,
   cidade :  “São Paulo” ,
   estado :  “SP” ,
   cep :  “12345”
}
```

O melhor modelo seria incorporar ambos em um único documento:

```
{
   _id :  “jose” ,
   nome :  “José da Silva” ,
   endereço :  {
```

```
                rua :  “Duque de caxias” ,
                cidade :  “São Paulo” ,
                estado :  “SP” ,
                cep :  “12345”
            }
}
```

Em geral, a incorporação fornece melhor desempenho para operações de leitura, bem como a capacidade de solicitar e recuperar dados relacionados em uma única operação de banco de dados. Modelos de dados incorporados tornam possível atualizar dados relacionados em uma única operação de gravação atômica.

Modelos de dados normalizados descrevem relacionamentos usando referências entre documentos. Normalmente, use modelos de dados normalizados:

- Quando a incorporação resultaria na duplicação de dados, mas não forneceria vantagens suficientes de desempenho de leitura para compensar as implicações da duplicação.
- Para representar relacionamentos muitos-para-muitos mais complexos.
- Modelar grandes conjuntos de dados hierárquicos.

Tabela 4. Relacionamento entre documentos.

{ _id : <**ObjectId1**>, nome : “Jose da Silva” }	{ _id : <ObjectId2> , user_id : <**ObjectId1**>, endereço : { rua : “Duque de caxias” , cidade : “São Paulo” , estado : “SP” , cep : “12345” } }
	{ _id : <ObjectId3> , user_id : <**ObjectId1**>, acesso : { nivel : 3 , grupo : “Alunos” , } }

Fonte: O autor.

CAPÍTULO 4
Banco de Dados NoSQL – MongoDB

O MongoDB é um banco de dados distribuído, baseado em documentos e de propósito geral, que utiliza padrões NoSQL. No MongoDB, uma operação de gravação é atômica no nível de um único documento, mesmo que a operação modifique vários documentos incorporados em um único documento. É formado por um modelo de dados que não utiliza mecanismos de normalização e combina todos os dados relacionados em um único documento, em vez de normalizar em vários documentos e coleções.

A decisão-chave no design de modelos de dados para aplicativos MongoDB gira em torno da estrutura dos documentos e de como o aplicativo representa os relacionamentos entre os dados. O MongoDB permite que dados relacionados sejam incorporados em um único documento.

Para instalar o MongoDB no Windows, vá ao site do fabricante, download center na aba *Community Server* e escolha a versão de seu sistema operacional. No Windows, a instalação se dará na forma do "*next, next, finish*".

https://www.mongodb.com/download-center?ct=false#community.

No Linux, a instalação é um pouco mais prática, bastando apenas utilizar o *apt-get* ou qualquer outro gerenciador para efetuar a instalação. O *apt-get* é uma ferramenta de linha de comando que ajuda no manuseio de pacotes no Linux. Sua principal tarefa é recuperar as informações e pacotes das fontes autenticadas para instalação, atualização e remoção de pacotes junto com suas dependências. Para mais detalhes, digite *man apt* no terminal.

```
$ sudo apt-get install mongodb
```

Depois de tudo instalado, será disponibilizado um *daemon*. Um *daemon* é um programa Linux que é executado em segundo plano. Quase todos os *daemons* têm nomes que terminam com a letra "d". Por exemplo, *mongod* é o *daemon* que lida com o servidor MongoDB. O Linux geralmente inicia *daemons* no momento da inicialização. Os *scripts* de *shell* armazenados no diretório /etc/init.d são usados para iniciar e parar *daemons* basta iniciar o servidor pela linha de comando:

```
$ sudo mongod
```

Para visualizar se o *daemon* está sendo executado, digite:

```
$sudo service --status-all
```

Para iniciar o MongoDb no Sistema Operacional Windows, abra um terminal e digite:

```
C:\Program Files\MongoDB\Server\4.2\bin>mongod
```

Se tudo funcionou normalmente, a seguinte saída aparecerá no terminal Linux:

```
Automatically disabling TLS 1.0, to force-enable TLS 1.0
specify --sslDisabledProtocols 'none'
MongoDB starting : pid=12456 port=27017 dbpath=C:\data\
db\ 64-bit host=DESKTOP-JFCU43H
 targetMinOS: Windows 7/Windows Server 2008 R2
 db version v4.2.2
 git version: a0bbbff6ada159e19298d37946ac8dc4b497eadf
 allocator: tcmalloc
 modules: enterprise
 build environment:
     distmod: windows-64
     distarch: x86_64
     target_arch: x86_64
```

Entretanto, alguns erros poderão acontecer quando for iniciar o MongoDB. Um deles é por não encontrar o diretório "/data/db" ou o Socket está em uso. Para isso, crie o diretório com o comando:

```
sudo mkdir /data && sudo mkdir /data/db
```

Em seguida, ao tentar iniciar novamente no Linux, poderá aparecer a seguinte mensagem:

```
Failed to set up listener: SocketException: Address already
in use
```

Esse problema aconteceu porque, mesmo com erro, o *mongod* já reservou a porta para sua utilização. Para solucionar esse problema, é preciso parar o *daemon* e rodar novamente. Para isso, rode o seguinte comando e inicie novamente o MongoDB:

```
 $sudo killall 15 mongodb && sudo mongod
```

As conexões estarão prontas para serem feitas pela porta 27017. Podemos testar se tudo está funcionando e criar a base de dados pela linha de comando. Para se conectar ao SGDB, digite o comando no terminal Linux:

$sudo mongo

Ou no Windows:

```
C:\Program Files\MongoDB\Server\4.2\bin>mongo
```

Se tudo funcionou bem, a seguinte mensagem aparecerá no terminal:

```
MongoDB shell version v4.2.2
connecting to: mongodb://127.0.0.1:27017/?compressors=di
sabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("80ac6d7f-2f6b-
4dbd-8406-52eef76d0ca4") }
MongoDB server version: 4.2.2
```

Criar a base de dados

Quando o MongoDB estiver rodando, é necessário se conectar com um cliente ou uma aplicação, como fizemos anteriormente. Para listar quais bancos de dados estão criados em no SGBD, digite:

```
MongoDB Enterprise > show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
```

Um banco de dados em MongoDb nada mais são do que uma coleção de documentos. Para criar a base de dados para o exemplo, digite ***use db_aula.***

```
MongoDB Enterprise > use db_aula
switched to db db_aula
MongoDB Enterprise > db
db_aula
MongoDB Enterprise >
```

Se for digitado *show bds* nesse momento, a base de dados db_aula não aparecerá. Ela só será criada efetivamente após ser criada uma *collection*.

Para alternar entre as bases de dados existentes, também pode ser utilizado o comando *use*.

O MongoDB armazena documentos em coleções que são análogas às tabelas em bancos de dados relacionais, entretanto, não tem a estrutura de linhas e colunas, como vimos no capítulo 3.

Para criar uma *collection,* digite:

```
MongoDB Enterprise > db.createCollection('Alunos')
{ "ok" : 1 }
MongoDB Enterprise > show collections
Alunos
MongoDB Enterprise >
```

Para inserir um documento no MongoDB via linha de comando, digite:

```
MongoDB Enterprise > db.Alunos.save({nome:"Joao da
Silva", cpf:"123456", endereco:"Rua XV de Novembro,
123"})
WriteResult({ "nInserted" : 1 })
```

Perceba que o formato de dados inseridos é padrão JSON:

```
(
     {
     nome:"Joao da Silva",
     cpf:"123456",
     endereco:"Rua XV de Novembro, 123"
     }
)
```

As *collections* possuem métodos que podem ser executados. Esses métodos poderão ser acessados pela documentação, através do link: https://docs.mongodb.com/manual/core/databases-and-collections/.

As operações básicas de CRUD (*Create, Retrieve, Update, Delete*) podem ser acessadas pelo link: https://docs.mongodb.com/manual/crud/>.

Para a operação de criação, existem as seguintes opções:

» db.collection.insertOne()

» db.collection.insertMany()

Insira alguns documentos para testar no próximo exemplo, a consulta.

As operações de leitura recuperam (*retrieves*) documentos de uma coleção; ou seja, consulta uma coleção de documentos. O MongoDB fornece os seguintes métodos para ler documentos de uma coleção:

» db.collection.find()

```
MongoDB Enterprise > db.Alunos.find()
{ "_id" : ObjectId("5dfe7e2a83c58f651dded53a"), "nome" :
"Joao da Silva", "cpf" : "123456", "endereco" : "Rua XV de
Novembro, 123" }
{ "_id" : ObjectId("5dfe83ce83c58f651dded53b"), "nome" :
"Maria Oliveira", "cpf" : "6666666", "endereco" : "Rua
Principal, 777" }
```

Perceba que quando existe mais de um documento, eles serão identificados com o atributo ***ObjectId***.

Na operação de leitura, é possível utilizar filtros:

```
MongoDB Enterprise > db.Alunos.find({cpf:"123456"})
{ "_id" : ObjectId("5dfe7e2a83c58f651dded53a"), "nome" :
"Joao da Silva", "cpf" : "123456", "endereco" : "Rua XV de
Novembro, 123" }
```

Os operadores de consulta auxiliam como parâmetros:

Tabela 5. Lista de parâmetros de consulta.

$eq	Corresponde a valores iguais a um valor especificado.
$gt	Corresponde a valores maiores que um valor especificado.
$gte	Corresponde a valores maiores ou iguais a um valor especificado.
$in	Corresponde a qualquer um dos valores especificados em uma matriz.
$lt	Corresponde a valores inferiores a um valor especificado.
$lte	Corresponde a valores menores ou iguais a um valor especificado.
$ne	Corresponde a todos os valores que não são iguais a um valor especificado.
$nin	Corresponde a nenhum dos valores especificados em uma matriz.

Fonte: https://docs.mongodb.com/manual/reference/operator/query/.

As operações de atualização modificam documentos existentes em uma coleção. O MongoDB fornece os seguintes métodos para atualizar documentos de uma coleção:

» db.collection.updateOne()

» db.collection.updateMany()

» db.collection.replaceOne()

No MongoDB, as operações de atualização têm como alvo uma única coleção. Todas as operações de gravação no MongoDB são atômicas no nível de um único documento, podendo, então, especificar critérios ou filtros que identificam os documentos a serem atualizados. Esses filtros usam a mesma sintaxe que as operações de leitura.

Para alterar a *collection*, digite o seguinte comando:

```
MongoDB Enterprise > db.Alunos.updateMany({cpf:{$eq:
"123456"}},{$set:{cpf:"123455"}})
```

Onde:

```
db.Alunos.updateMany(     → Chamada de comanda na collection
{cpf:{$eq: "123456"}},    → Filtros
{$set:{cpf:"123455"}})    → Ação de alteração
```

Logo na sequência, será informado que houve alteração:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount"
: 1 }
```

Tente pesquisar pelo registro antigo e pela nova alteração:

```
MongoDB Enterprise > db.Alunos.find({cpf:"123456"})
MongoDB Enterprise > db.Alunos.find({cpf:"123455"})
{ "_id" : ObjectId("5dfe7e2a83c58f651dded53a"), "nome" :
"Joao da Silva", "cpf" : "123455", "endereco" : "Rua XV de
Novembro, 123" }
```

As operações de exclusão removem documentos de uma coleção. O MongoDB fornece os seguintes métodos para excluir documentos de uma coleção:

- db.collection.deleteOne()
- db.collection.deleteMany()

No MongoDB, as operações de exclusão têm como destino uma única coleção. Todas as operações de gravação no MongoDB são atômicas no nível de um único documento. É possível especificar critérios ou filtros que identificam os documentos a serem removidos. Esses filtros usam a mesma sintaxe que as operações de leitura.

```
MongoDB Enterprise > db.Alunos.deleteMany({cpf:"123455"})
{ "acknowledged" : true, "deletedCount" : 1 }
```

Quando uma única operação de gravação (por exemplo, db.collection.updateMany()) modifica vários documentos, a modificação de cada documento é atômica, mas a operação como um todo não é atômica. Ao executar operações de gravação de documentos múltiplos, seja através de uma única operação de gravação ou de várias operações de gravação, outras operações podem se intercalar.

Para situações que exigem atomicidade de leituras e gravações em vários documentos (em uma ou várias coleções), o MongoDB suporta transações com vários documentos:

- MongoDB suporta transações com vários documentos em conjuntos de réplicas.
- A MongoDB introduz transações distribuídas, que adicionam suporte para transações com vários documentos em *clusters* fragmentados e incorporam o suporte existente para transações com vários documentos em conjuntos de réplicas.

BANCOS DE DADOS PARA BIG DATA - HADOOP

UNIDADE III

CAPÍTULO 1

Introdução

O projeto *Hadoop* é um *framework*, ou seja, um conjunto de *softwares* para computação confiável, escalável e distribuída. Ele possui uma estrutura projetada para processar um grande volume de dados de forma distribuída e projetado escalonar de um simples servidor individual a milhares de máquinas. Sua utilização também ganha muitos adeptos devido ao grande número de bibliotecas e *software* de terceiros, que realizam tarefas essenciais para o trabalho dos cientistas de dados, além, inclusive, possuir mecanismos que detectam e recuperam falhas.

História do *Hadoop*

O *Hadoop* foi criado pelos cientistas da computação Doug Cutting e Mike Cafarella, inicialmente para dar suporte ao processamento no mecanismo de pesquisa de código aberto Apache Nutch (https://nutch.apache.org/) e no rastreador da Web. Depois que o Google publicou documentos técnicos detalhando sua estrutura de programação do Google File System e do MapReduce em 2003 e 2004, Cutting e Cafarella modificaram os planos tecnológicos anteriores e desenvolveram uma implementação MapReduce baseada em Java e um sistema de arquivos modelado no Google. Ao mesmo tempo, Cutting foi contratado pela empresa de serviços de Internet Yahoo, que se tornou o primeiro usuário de produção do *Hadoop* no final de 2006.

O filho de Doug Cutting nomeou um de seus brinquedos de *Hadoop*, um elefante amarelo. Doug então usou o nome para seu projeto de código aberto, porque era fácil soletrar, pronunciar e não ser usado em outros lugares.

O uso da estrutura cresceu nos próximos anos e foram criados três fornecedores independentes do *Hadoop*:

- Cloudera, em 2008 (https://www.cloudera.com/products/open-source/apache-hadoop.html).
- MapR Technologies, um ano depois (https://mapr.com/).
- AWS lançou um serviço de nuvem *Hadoop* chamado *Elastic MapReduce,* em 2009 (https://aws.amazon.com/pt/emr/).

Isso aconteceu antes de a Apache Foundation lançar sua primeira versão do *Hadoop*, disponibilizada em dezembro de 2011.

A natureza flexível de um sistema *Hadoop* significa que as empresas podem adicionar ou modificar funcionalidades conforme suas necessidades. Tanto uma grande corporação quanto apenas uma pessoa física são livres para alterá-lo para seus próprios propósitos. As modificações feitas no *software* por engenheiros especialistas, por exemplo, Amazon e Google, são retornadas à comunidade de desenvolvimento, onde são frequentemente usados para melhorar o produto "oficial". Essa forma de desenvolvimento colaborativo entre usuários voluntários e comerciais é uma característica essencial do *software* de código aberto, ou Open Source.

Ele é mantido pela *Apache Foudation* e, como a maioria dos projetos da organização, é composto de vários módulos e projetos, que inclui:

- ***Hadoop* Common**: os utilitários comuns que oferecem suporte aos outros módulos do *Hadoop*.
- ***Hadoop* Distributed File System (HDFS)**: um sistema de arquivos distribuídos que fornece acesso de alta taxa de transferência aos dados do aplicativo.
- ***Hadoop* YARN**: uma estrutura para agendamento de tarefas e gerenciamento de recursos de *cluster*.
- ***Hadoop* MapReduce:** um sistema baseado em YARN para processamento paralelo de grandes conjuntos de dados.

Algumas considerações devem ser levadas antes de escolher um banco de dados para um projeto de *Big Data*. Embora esse material aborde alguns, existem vários outros e cada um para um determinado tipo de aplicação e, muitas vezes, trabalhando juntos. Aqui abordamos o *Hadoop* e o MongoDB e suas particularidades. A seguir, foi elaborada uma lista com as principais diferenças entre os dois e cabe ao projetista avaliar a relevância sob o projeto:

» Linguagem de programação

› O *Hadoop* em Programação Java.

› MongoDB C ++.

» Código aberto

› Tanto o *Hadoop* quanto o MongoDB são de código aberto.

» Escalabilidade

› Tanto o *Hadoop* quanto o MongoDB são escaláveis.

» NoSQL

› O *Hadoop* não suporta NoSQL nativamente, apenas através de um outro *software* chamado HBase.

› O MongoDB suporta NoSQL.

» Dados Estruturados

› O *Hadoop* possui uma estrutura de dados flexível.

› O MongoDB suporta a estrutura de dados baseada em documentos.

» Custo

› *Hadoop* precisa implementar um ecossistema mais complexo.

› O MongoDB é um único *software*, portanto, mais econômico.

» Volumes de Dados

› O *Hadoop* pode lidar com grandes volumes de dados.

› O MongoDB pode lidar com o tamanho moderado de dados, menos que o *Hadoop*.

» Formato de dados geoespacial

› O *Hadoop* não pode lidar com dados geoespaciais com eficiência.

› O MongoDB pode analisar dados geoespaciais com sua capacidade de indexação.

CAPÍTULO 2
Conceitos de *Big Data* com *Hadoop*

Que o *Big Data* está surgindo como uma oportunidade para as organizações é incontestável e inevitável. O que as organizações precisam saber – e muitas delas já estão tirando proveito disso – são os muitos benefícios oferecidos pelo *Big Data* Analytics. Elas estão examinando grandes conjuntos de dados para descobrir todos os padrões ocultos, correlações desconhecidas, tendências de mercado, preferências do cliente e outras informações comerciais úteis.

Essas descobertas analíticas estão ajudando as organizações, por exemplo, em *marketing* mais eficaz, novas oportunidades de receita e melhor atendimento ao cliente. Eles estão melhorando a eficiência operacional, vantagens competitivas sobre organizações rivais e outros benefícios comerciais. Portanto, citaremos algumas dificuldades e problemas com diversas abordagens, para depois fundamentarmos soluções com o *Big Data Hadoop*.

Na abordagem tradicional, o principal problema era lidar com a heterogeneidade dos dados, isto é, estruturados, semiestruturados e não estruturados. Os RDBMS concentram-se principalmente em dados estruturados, como transações bancárias, dados operacionais etc. e o *Hadoop* é especializado em dados semiestruturados e não estruturados, como texto, vídeos, áudios, postagens no Facebook, registros etc. A tecnologia RDBMS é um sistema comprovado, altamente consistente e amadurecido suportado por muitas empresas. Por outro lado, o *Hadoop* está em demanda devido ao *Big Data*, que consiste principalmente de dados não estruturados em diferentes formatos.

Toda demanda parte de uma necessidade e, muitas vezes, essas trazem problemas a serem resolvidos. Particularmente no mundo *Big Data* não seria diferente, e serão listados alguns dos principais problemas enfrentados, em seguida, como resolvê-los utilizando *Hadoop*.

O primeiro e principal problema é a quantidade de dados que estão sendo gerados a cada segundo, o que não significa que deverão ser armazenados em um único local. Várias empresas captam e utilizam uma quantidade de dados inimagináveis. Por isso, armazenar esses dados nos meios tradicionais está impossível e os motivos são diversos, como banda de rede e armazenamento.

Não bastasse a quantidade dos dados, a qualidade e a heterogeneidade também devem apresentar muitos problemas. A qualidade deve ter um impacto um pouco menor, pois, atualmente, muitos tratamentos no *front* estão sendo feitos, justamente para que eles cheguem com uma certa tolerância e usabilidade. O problema maior está na diversidade dos dados que chegam.

Normalmente, os dados internos de uma empresa giram em torno dos usuários, trabalhando em um ERP, e-mails, pastas compartilhadas, redes sociais e aplicativos de mensagens. Imagine agora ter que fazer um trabalho de mineração de dados e reunir todos esses formatos de dados para que a informação faça sentido. Os dados são estruturados (base do ERP), semiestruturados (redes sociais) e não estruturados (pastas compartilhadas).

Quando é dito armazenamento, logo se imagina a facilidade em adquirir alguns HDs de Terabytes e colocar todos esses dados lá. Entretanto, os dados não bastam ser somente armazenados. Temos que lembrar que, na sequência, será feito um trabalho de mineração e valores serão extraídos. Se o resgate, as consultas ou buscas desses dados forem lentas, de nada adiantou armazená-los. A capacidade e a segurança do disco rígido estão aumentando, mas a velocidade de transferência e acesso não estão aumentando na mesma taxa de proporção.

Por exemplo, se um disco com uma taxa de transferência de 100 Mbps está processando 1 TB de dados, levará cerca de 2,91 horas para a conclusão. Se for aumentado o número de máquinas quatro vezes, por exemplo, para a mesma quantidade de dados, serão necessários aproximadamente 43 minutos. A tecnologia do disco também é um fator importante. Discos com tecnologia SSD também tendem a ser mais rápidos. Veja, por exemplo, a comparação de dois discos, instalados em PCs comuns:

Foi utilizado o *hwinfo* para obter as especificações:

Tabela 6. Informações sobre discos.

Hardware Class: disk	Hardware Class: disk
Model: "KINGSTON SA400S3"	Model: "VBOX HARDDISK"
Vendor: "KINGSTON"	Vendor: "VBOX"
Device: "SA400S3"	Device: "HARDDISK"
Device File: /dev/sda	Device File: /dev/sda
Size: 468862128 sectors a 512 bytes	Size: 20971520 sectors a 512 bytes
Capacity: 223 GB	Capacity: 10 GB

Fonte: O autor.

Depois, para testes de leitura foi utilizado o *hdparm*:

Tabela 7. Teste de velocidade de leitura.

sudo hdparm -I /dev/sda | grep -i speed	**sudo hdparm -I /dev/sda | grep -i speed**
* Gen1 signaling speed (1.5Gb/s)	* Gen2 signaling speed (3.0Gb/s)
* Gen2 signaling speed (3.0Gb/s)	
* Gen3 signaling speed (6.0Gb/s)	

Fonte: O autor.

E, em seguida, um teste para gravação com *dd*.

Tabela 8. Teste de velocidade de gravação.

sudo dd if=/dev/zero of=/tmp/output.img bs=8k count=10k; sudo rm -vf /tmp/output.img	**sudo dd if=/dev/zero of=/tmp/output.img bs=8k count=10k; sudo rm -vf /tmp/output.img**
10240+0 registros de entrada	10240+0 registros de entrada
10240+0 registros de saída	10240+0 registros de saída
83886080 bytes (84 MB, 80 MiB)	83886080 bytes (84 MB, 80 MiB)
copiados, **0,0495489 s, 1,7 GB/s**	copiados, **0,143656 s, 584 MB/s**
removido '/tmp/output.img'	removido '/tmp/output.img'

Fonte: O autor.

Outro componente da infraestrutura de um *cluster Hadoop* é a rede. Apesar do tráfego do *Hadoop* ser pequeno, ele pode congestionar redes lentas. Existem várias formas e aparelhos para testar o desempenho da rede, porém, se não dispor de nenhum desses produtos em mãos, o *ping* é um bom utilitário. Quando enviado pela rede com pacotes grandes, o tempo de resposta deverá ser muito menor que 1 milissegundo. O comando para esse teste é:

```
$ ping -s 64512 IP_destino
```

Portanto, a velocidade de acesso e processamento é o maior problema enfrentado pelo armazenamento de *Big Data*. Vejamos agora algumas possibilidades em que o *Hadoop* pode ajudar nesses problemas. Serão mencionados alguns componentes do *Hadoop* apenas para ilustrar os exemplos. Esses componentes serão detalhados nos próximos capítulos.

O problema sobre armazenamento poderá ser resolvido com o componente chamado HDFS, que fornece uma maneira distribuída de armazenar os dados em formato de blocos nos DataNodes, e quais poderão ser especificados o seu

tamanho. Exemplo: os dados que forem armazenados chegaram ao HD para serem escritos e possuir 512 MB e se o HDFS estiver configurado de tal forma, ele criará 128 MB de blocos de dados. Portanto, o HDFS dividirá os dados em 4 blocos (512/128) e os armazenará em diferentes DataNodes, além de fazer sua replicação em blocos diferentes.

O problema de dimensionamento também será resolvido devido à escala. O *Hadoop* se concentra sua escalada na horizontal e não na vertical, e isso só é possível por poder adicionar nodes (nós) a qualquer momento no *cluster* HDFS, em vez de aumentar os recursos de seus DataNodes. Por exemplo, se a demanda de dados for de 1TB, não será preciso ter um HD de 1TB, podendo distribuir em vários de 128GB.

Outro problema a se resolver com HDFS é a variedade dos dados. Como o HDFS não faz validação de esquemas, é possível armazenar todos os tipos de dados, sejam eles estruturados, semiestruturados ou não estruturados. Outra vantagem é a forma com que grava e recupera os dados. Ele escreve uma vez e lê vários modelos, ou seja, é possível escrever os dados apenas uma vez e lê-los várias vezes para encontrar informações.

O desafio principal em qualquer implementação de *Big Data* foi a velocidade. É utilizada a seguinte lógica: o processamento é movido para os dados e não os dados para o processamento. Isso significa que, em vez de os dados serem movidos para o nó principal, para serem processados, o processamento é enviado para os nós escravos, os dados são processados paralelamente e o resultado é enviado de volta ao cliente. Isso é possível através do componente YARN.

Embora tenha sido visto que o *Hadoop* tem um conjunto maior de componentes e funcionalidades, utilizadas inclusive pelo *Spark*, o que a torna uma ferramenta robusta e poderosa, existem algumas desvantagens em sua implementação e utilização:

- Baixa velocidade de processamento

  No *Hadoop*, o algoritmo MapReduce, que é paralelo e distribuído, processa conjuntos de dados realmente grandes. Estas são as tarefas que precisam ser executadas aqui:

  - **Mapa**: o mapa pega uma quantidade de dados como entrada e os converte em outro conjunto de dados, que novamente é dividido em pares chave / valor.

- **Reduzir**: a saída da tarefa "mapa" é inserida em "reduzir" como entrada. Na tarefa "reduzir", como o nome sugere, esses pares de chave / valor são combinados em um conjunto menor de tuplas. A tarefa "reduzir" é sempre realizada após o mapeamento.

» Processamento em lote

O *Hadoop* implanta o processamento em lote, que coleta os dados e depois os processa em massa posteriormente. Embora o processamento em lote seja eficiente para processar grandes volumes de dados, ele não processa dados em fluxo. Por isso, o desempenho é menor.

» **Sem *pipelining* de dados**

O *Hadoop* não oferece suporte ao *pipelining* de dados (ou seja, uma sequência de estágios em que o ID de saída do estágio anterior é a entrada do próximo estágio).

» Não é fácil de usar

Os desenvolvedores do MapReduce precisam escrever seu próprio código para cada operação, o que dificulta muito o trabalho. E também, o MapReduce não tem modo interativo.

» Latência

No *Hadoop*, a estrutura do MapReduce é mais lenta, pois suporta diferentes formatos, estruturas e grandes volumes de dados.

» Linha de código longa

Como o *Hadoop* é escrito em Java, o código é longo. E isso leva mais tempo para executar o programa.

Em um artigo denominado "*Top Advantages and Disadvantages of Hadoop 3*", Data Flair (2019), listou as principais vantagens e desvantagens que encontraram em utilização e implementações do *Hadoop*:

Vantagens

» **Fontes de dados variadas:** o *Hadoop* aceita uma variedade de dados.

» **Custo-benefício:** o *Hadoop* é uma solução econômica, pois usa um *cluster* de *hardware* comum para armazenar dados.

» **Desempenho:** o *Hadoop*, com seu processamento distribuído e arquitetura de armazenamento distribuído, processa grandes quantidades de dados com alta velocidade.

» **Tolerante a falhas:** no *Hadoop* 3.0, a tolerância a falhas é fornecida pela codificação de apagamento.

» **Altamente disponível:** no *Hadoop* 2.x, a arquitetura HDFS tem um único NameNode ativo e um único NameNode em espera.

» **Baixo tráfego de rede:** no *Hadoop*, cada tarefa enviada pelo usuário é dividida em várias subtarefas.

» **Alto rendimento:** *Hadoop* armazena dados de maneira distribuída, o que permite usar o processamento distribuído com facilidade.

» **Código aberto:** podemos modificar o código fonte para atender a um requisito específico.

» **Escalável:** o *Hadoop* trabalha com o princípio da escalabilidade horizontal, nós podem ser adicionados ao *cluster* do *Hadoop* rapidamente, tornando-o uma estrutura escalável.

» **Facilidade de uso:** a estrutura do *Hadoop* cuida do processamento paralelo, os programadores do MapReduce não precisam cuidar do alcance do processamento distribuído, isso é feito automaticamente no *back-end*.

» **Compatibilidade**: a maior parte da tecnologia emergente do *Big Data* é compatível com o *Hadoop* como *Spark*, Flink etc.

» **Várias linguagens de programação suportadas:** os desenvolvedores podem codificar usando muitas linguagens no *Hadoop*, como C, C ++, Perl, Python.

Desvantagens do *Hadoop*

» **Problema com arquivos pequenos:** o *Hadoop* é adequado para um pequeno número de arquivos grandes, mas quando se trata do aplicativo que lida com um grande número de arquivos pequenos, o *Hadoop* falha aqui.

- **Vulnerável por natureza:** o *Hadoop* é escrito em Java, que é uma linguagem de programação amplamente usada; portanto, é facilmente explorado por cibercriminosos.
- **Processamento de custos indiretos:** no *Hadoop*, os dados são lidos e gravados no disco, o que torna as operações de leitura/gravação muito caras quando lidamos com tera e petabytes de dados. O *Hadoop* não pode fazer cálculos na memória, portanto, incorre em sobrecarga de processamento.
- **Suporta apenas processamento em lote:** no núcleo, o *Hadoop* possui um mecanismo de processamento em lote que não é eficiente no processamento de fluxo.
- **Processamento Iterativo:** o *Hadoop* não pode executar o processamento iterativo por si só.
- **Segurança:** por segurança, o *Hadoop* usa a autenticação Kerberos, que é difícil de gerenciar. Falta criptografia nos níveis de armazenamento e rede, o que é um grande ponto de preocupação.

CAPÍTULO 3

Arquitetura Principal do *Hadoop*

Hadoop Distributed File System

O *Hadoop Distributed File System* é um sistema de arquivos distribuído que permite o armazenamento de grande volume de dados de maneira tolerante a falhas. Sistemas de arquivos são estruturas lógicas que os sistemas operacionais, como o Linux, utilizam para gerenciar os arquivos. Quanto mais os dados vão aumentando, mais complexo fica o gerenciamento dos arquivos pelos servidores. O gerenciamento de arquivos pelo sistema operacional inclui desde manipulação, como gravação e leitura, até controle de nível de acesso e segurança. Entretanto, esse gerenciamento só poderá ser efetuado em uma única máquina. Já o HDFS também efetua todo esse gerenciamento, porém em um sistema distribuído de computação, se comportando como se fosse uma única máquina em um sistema de cluster.

Existem três componentes principais do *Hadoop* HDFS:

» DataNodes

O HDFS armazena os dados reais e de maneira distribuída nos DataNodes, dividindo os variados tipos de arquivos de entrada em vários blocos.

A lógica do DataNode é a seguinte:

› Quando inicializa, o DataNode faz uma validação com o nó principal, chamado NameNode.

› Ele verifica o ID do espaço de alocação e a versão do *software* do DataNode.

› Também, envia uma lista com as características de bloco para o NameNode a cada três segundos, informando que está ativo.

» NameNode

NameNode é o *daemon* principal que mantém e gerencia os DataNodes. No caso de falha do DataNode, o NameNode escolhe novos DataNodes para novas réplicas, equilibra o uso do disco

e gerencia o tráfego de comunicação para os DataNodes. Ele é responsável por gerenciar o espaço de nome do sistema de arquivos, controlar tarefas de manipulação de arquivos e diretórios, como: abrir, fechar, nomear e renomear, anexar dados ao arquivo, controlar o nível de acesso dos usuários, como permissão de arquivo, cota de disco, registro de data e hora da modificação, horário de acesso e os logs, grava alterações incrementais, como renomear o arquivo. Essas informações ficam contidas em um arquivo chamado *FSImage*. Todas as modificações feitas no *FSImage* são armazenadas em um arquivo chamado *EditLogs*.

» Secondary NameNode

O Secondary NameNode pode ser pensado como um assistente do NameNode que leva parte da carga de trabalho do NameNode. Caso o NameNode não seja reiniciado de tempos em tempos, o tamanho de seu log aumenta e atrasa a reinicialização do *cluster*. O papel do Secundary NameNode aplica atualizações no FSImage em intervalos e atualiza o novo FSImage no NameNode primário.

Hadoop MapReduce

O objetivo do HDFS é fazer a gestão dos enormes volumes de dados em um sistema distribuídos de armazenamento, ou seja, em vários *clusters* de computadores. Uma vez armazenados, esses dados precisam ser processados, ou seja, deverão ser extraídos para um conjunto de informações com alguma utilidade. Para isso, foi criado outro componente: o MapReduce.

MapReduce é um conceito de redução e otimização dos dados, já armazenados, existentes antes do *Hadoop*, que foi incluído pela equipe de desenvolvimento para dentro do *framework*. MapReduce é o componente de processamento de dados do *Hadoop*. Aplica a computação em conjuntos de dados em paralelo, melhorando, assim, o desempenho. A estrutura MapReduce opera exclusivamente em pares, ou seja, a estrutura exibe a entrada para o trabalho como um conjunto de pares e produz um conjunto de pares como a saída do trabalho, possivelmente de diferentes tipos.

Ele funciona em duas fases:

» **Map** – como o nome sugere, ele faz apenas o mapeamento, ou seja, recebe a entrada como pares de valores-chave e produz

saída como pares de valores-chave, processa os dados e manda para a fase de redução.

» **Reduce** – classifica o par de valores-chave e aplica o tipo de resumo dos cálculos aos pares de valores-chave, reduzindo o seu conteúdo.

A lógica é a seguinte:

» Um mecanismo chamado Mapper lê o bloco de dados e o converte em pares de valores-chave.

» Os pares de chave entram no redutor.

» O redutor recebe tuplas de dados de vários Mappers.

» O redutor aplica agregação a essas tuplas com base na chave.

» A saída final do redutor é gravada no HDFS.

O MapReduce ainda cuida de possíveis falhas e recuperação dos dados nos nós. Sua documentação oficial.

Hadoop YARN

É através do YARN (*Yet Another Resource Manager*) que o *Hadoop* monitora e gerencia os recursos. Ele lida com as cargas de trabalho como processamento de fluxo, processamento interativo e processamento em lote em uma única plataforma e tem dois componentes principais: **ResourceManager** e o **NodeManager**. Juntos, eles formam a estrutura de computação de dados, na qual o ResourceManager é quem gerencia todos os recursos dos aplicativos no sistema e o NodeManager é o agente responsável pelos contêineres, monitorando o uso de recursos (CPU, memória, disco, rede) e relatando o mesmo ao Scheduler do ResourceManager.

Segundo a documentação do *Hadoop* YARN: "a ideia fundamental do YARN é dividir as funcionalidades de gerenciamento de recursos e agendamento / monitoramento de tarefas em daemons separados. A ideia é ter um **ResourceManager** global e um **ApplicationMaster** por aplicativo". O ApplicationMaster por aplicativo é, na verdade, uma biblioteca específica da estrutura e recebe a tarefa de negociar recursos do ResourceManager e trabalhar com o NodeManager para executar e monitorar as tarefas. O ResourceManager possui dois componentes principais: o **Scheduler** e **ApplicationsManager**.

Scheduler é responsável pela alocação de recursos para os vários aplicativos em execução, capaz de gerenciar filas, não executa nenhum monitoramento e agenda tarefas baseado em requisitos de recursos de aplicativos. Ele ainda possui *plug-ins* que são desenvolvidos conforme a ferramenta vai evoluindo, podendo citar o **CapacityScheduler** e o **FairScheduler**.

ApplicationsManager é responsável por aceitar os envios de tarefas, negociando o primeiro contêiner para executar o ApplicationMaster específico do aplicativo e fornece o serviço para reiniciar o contêiner ApplicationMaster em caso de falha. O ApplicationMaster por aplicativo tem a responsabilidade de negociar contêineres de recursos apropriados do Scheduler, rastreando seu status e monitorando o progresso.

CAPÍTULO 4
Ecossistema *Hadoop*

O ecossistema do *Hadoop* é um conjunto de *software* produzido por terceiros em parceria com a Apache Foundation, que pode ser instalado para suprir as necessidades de projetos específicos.

Hive

O Hive é um projeto de **Data Warehouse** construído na parte superior do Apache *Hadoop* que facilita a leitura, gravação e gerenciamento de grandes conjuntos de dados que residem no armazenamento distribuído usando SQL. Ele possui a linguagem própria, chamada HQL ou **Hive Query Language**.

As principais partes da *Hive* são:

» ***MetaStore*** – armazena metadados.

» ***Driver*** – gerencia o ciclo de vida da instrução HQL.

» ***Query Compiler*** – analisa a sintaxe da HQL.

» ***Hive Server*** – fornece interface para o servidor JDBC/ODBC.

O Hive fornece muitas funções predefinidas para análise, ou o desenvolvedor também pode definir suas próprias funções personalizadas, chamadas UDFs ou funções definidas pelo usuário. É possível ver todas as suas funções na documentação, disponível nas referências.

PIG

Pig é uma linguagem semelhante ao SQL usada para consultar e analisar dados armazenados no HDFS que utiliza a língua latina própria. Sua função é simples: carrega os dados, aplica um filtro e despeja os dados no formato necessário. Alguns recursos do PIG:

» **Extensibilidade** – possibilidade de criar as próprias funções.

» **Oportunidades de otimização** – o Pig otimiza automaticamente a consulta.

» **Lida com todos os tipos de dados** – o Pig analisa a estrutura enquanto executa consultas.

É possível ver todas as suas funções na documentação, disponível nas referências.

HBase

O HBase é um sistema gerenciador de banco de dados NoSQL, orientado a colunas, de código aberto, não relacional, criado na parte superior do HDFS e fornece acesso de leitura / gravação em tempo real a grandes conjuntos de dados.

No HBase, os dados são armazenados em tabelas, que possuem linhas e colunas, porém não é certo compará-lo aos bancos de dados relacionais. Uma tabela HBase pode se comparar a um mapa multidimensional.

Uma tabela HBase consiste em várias linhas, uma linha no HBase consiste em uma chave de linha e uma ou mais colunas com valores associados a elas. Uma coluna no HBase consiste em uma família de colunas e um qualificador de coluna, delimitados por um caractere de dois pontos ":". O design do HBase contém muitas tabelas. Cada uma dessas tabelas deve ter uma chave primária. As tentativas de acesso às tabelas HBase usam essa chave primária.

No entanto, o HBase possui muitos recursos que suportam escala linear e modular. Os *clusters* HBase se expandem adicionando **RegionServers** hospedados em servidores de classe de *commodity*. O **RegionServer** é um processo que lida, grava, atualiza e exclui solicitações de clientes. Ele é executado em todos os nós em um *cluster* do *Hadoop* que é o HDFS DataNode. Um RDBMS pode escalar bem, mas apenas até certo ponto – especificamente, o tamanho de um único servidor de banco de dados – e para melhor desempenho requer *hardware* e dispositivos de armazenamento especializados.

Alguns recursos do HBase são:

» Leituras/gravações fortemente consistentes.

» Compartilhamento automático de tabelas distribuídas.

» Controle de falha automático do **RegionServer.**

» Integração *Hadoop* HDFS.

» Suporte a MapReduce.

» API do cliente Java.

» API REST.

Nem todos os problemas são solucionados com o HBase. Ele não é adequado para todos os problemas. Existem situações nas quais ele pode mais atrapalhar do que ajudar. Por exemplo, se for utilizar HBase, é preciso ter a certeza que possui dados suficientes, ou seja, se possui centenas de bilhões de linhas de dados, ele é a ferramenta ideal. Se não, é recomendável a utilização de um SGDB tradicional de mercado, como PostgreSQL, MySql, SQLServer etc.

Caso precise mudar de tecnologia de um banco de dados tradicional para um HBase, tenha certeza que a lógica de negócios, como consultas e processamentos, esteja na aplicação. Caso contrário, toda a lógica precisará ser reescrita, pois não existe portabilidade.

Por fim, verifique se possui um bom *hardware*, pois o HBase pode funcionar bem em ambientes de desenvolvimento, mas pode ser problemático em produção.

É possível ver todas as suas funções na documentação, disponível nas referências.

Mahout

O *Mahout* fornece um ambiente para a criação de aplicativos de aprendizado de máquina escaláveis. Com os algoritmos de aprendizado de máquina, é possível construir máquinas de autoaprendizado que "aprendem" por si mesmas sem serem explicitamente programadas. Por exemplo, baseado no comportamento do usuário, padrões de dados e experiências passadas, decisões poderão ser tomadas através de previsões futuras feitas pelas máquinas. O *Mahout* realiza filtragem, agrupamento e classificação colaborativos. Algumas pessoas também consideram a falta de conjuntos de itens frequentes como função de *Mahout*. Suas principais funções são:

» **Filtragem colaborativa:** o *Mahout* analisa o comportamento do usuário e faz recomendações.

» **Agrupamentos**: organiza um grupo semelhante de dados.

» **Classificação**: classifica e categoriza dados em vários subgrupos.

» **Conjunto de itens frequentes ausentes**: verifica quais objetos provavelmente aparecerão juntos e faz sugestões, se estiverem ausentes.

Em sua linha de comando, é possível chamar vários algoritmos, possui um conjunto de bibliotecas e algoritmos prontos baseados em casos de uso.

É possível ver todas as suas funções na documentação, disponível nas referências.

ZooKeeper

O *ZooKeeper* é um serviço centralizado para manter informações de configuração, nomear, fornecer sincronização distribuída e fornecer serviços de grupo. Após sua implementação, o *Hadoop* enfrenta muitos problemas de concorrência por recursos, entre as tarefas. Se forem corrigidos, *deadlocks* frequentes poderiam ocorrer. Mesmo quando feitas corretamente, diferentes implementações desses serviços levam à complexidade do gerenciamento quando os aplicativos são implantados. Ele também trabalha na serialização sempre quando as tarefas executam duas ou mais tarefas por vez.

É possível ver todas as suas funções na documentação, disponível nas referências.

Oozie

Oozie é um aplicativo da Web de código aberto escrito em Java. É um sistema agendador de fluxo de trabalho para gerenciar tarefas do *Hadoop* MapReduce, Pig, Hive e *Sqoop*, além de combinar vários trabalhos em uma única unidade de trabalho, sendo escalável e pode gerenciar milhares de fluxos de trabalho em um *cluster Hadoop*. *Oozie* trabalha criando fluxo de trabalho, podendo iniciar, parar, suspender e executar novamente tarefas com falha. Ele armazena e executa um fluxo de trabalho composto por tarefas do *Hadoop*. Ele armazena o trabalho como Gráfico Acíclico Dirigido para determinar a sequência de ações que serão executadas e executa trabalhos de fluxo de trabalho com base em agendamentos predefinidos e disponibilidade de dados.

É possível ver todas as suas funções na documentação, disponível nas referências.

Sqoop

O *Sqoop* importa dados em massa, de fontes externas, como dados estruturados de SGDBs, para componentes compatíveis do ecossistema *Hadoop* e também faz o processo inverso, transfere dados do *Hadoop* para outras fontes externas. O *Sqoop* pode lidar com dados estruturados e não estruturados.

É possível ver todas as suas funções na documentação, disponível nas referências.

Flume

O *Flume* é um serviço distribuído para coletar, agregar e mover com eficiência grandes quantidades de dados de log, possui uma arquitetura simples e flexível, baseada no fluxo de dados de *streaming*, é robusto e tolerante a falhas, com mecanismos de confiabilidade ajustáveis e muitos mecanismos de ***failover*** e recuperação. Ele usa um modelo de dados extensível simples que permite aplicativos analíticos online.

É possível ver todas as suas funções na documentação, disponível nas referências.

Ambari

O projeto *Apache Ambari* visa a facilitar o gerenciamento do *Hadoop*, desenvolvendo *software* para provisionar, gerenciar e monitorar *clusters* do Apache *Hadoop*, fornecendo uma interface intuitiva, na Web e API REST.

Seguem as funções que os administradores têm disponíveis no *Ambari*:

- Provisionar um *cluster Hadoop*.
- Assistente passo a passo para instalar os serviços do *Hadoop*.
- Manipula a configuração dos serviços *Hadoop* para o *cluster*.
- Fornece gerenciamento central para iniciar, parar e reconfigurar os serviços do *Hadoop* em todo o *cluster*.
- Fornece um painel para monitorar a integridade e o status do *cluster Hadoop*.
- Permite que desenvolvedores de aplicativos e integradores de sistemas, através das APIs REST.

É possível ver todas as suas funções na documentação, disponível nas referências.

Apache Drill

O *Apache Drill* é um mecanismo de consulta SQL sem esquema. Ele suporta uma variedade de bancos de dados e sistemas de arquivos NoSQL, incluindo HBase, MongoDB, HDFS, NAS e arquivos locais. Uma única consulta pode unir dados de vários armazenamentos de dados. O otimizador com reconhecimento

de armazenamento de dados da *Drill* reestrutura automaticamente um plano de consulta para aproveitar os recursos de processamento interno do armazenamento de dados.

É possível ver todas as suas funções na documentação, disponível nas referências.

Solr

O *Solr* é altamente confiável, escalável e tolerante a falhas, fornecendo indexação distribuída, replicação e consultas com balanceamento de carga, *failover* e recuperação automatizados, configuração centralizada e muito mais. O *Solr* fornece recursos de pesquisa e navegação para muitos dos maiores sites da Internet no mundo.

Várias características do *Solr* são as seguintes:

- Altamente escalável, confiável e tolerante a falhas.
- Fornece indexação distribuída, *failover* e recuperação automatizados, consulta com balanceamento de carga, configuração centralizada e muito mais.
- Você pode consultar o *Solr* usando HTTP GET e receber o resultado em JSON, binário, CSV e XML.
- O *Solr* fornece recursos correspondentes, como frases, curingas, agrupamento, junção e muito mais.
- Ele é enviado com uma interface administrativa integrada, permitindo o gerenciamento de instâncias de *Solr*.
- *Solr* tira proveito da indexação quase em tempo real da Lucene. Ele permite que você veja o seu conteúdo quando quiser.

É possível ver todas as suas funções na documentação, disponível nas referências.

BANCO DE DADOS PARA *BIG DATA – SPARK*

UNIDADE IV

CAPÍTULO 1

Introdução

O Apache *Spark* é um sistema de computação em *cluster* rápido e de uso geral, além de ser um *framework* para processamento de dados em larga escala. Ele fornece APIs de alto nível em várias linguagens de programação, como Java, Scala, Python e R, e um mecanismo otimizado que suporta gráficos de execução geral que integra basicamente todas as ferramentas de *Big Data*.

Ele também suporta um rico conjunto de ferramentas, incluindo *Spark* SQL para SQL e processamento de dados estruturados, *MLlib* para aprendizado de máquina, *GraphX* para processamento de gráficos e *Spark Streaming*. O *Spark* é um mecanismo de processamento de dados desenvolvido para fornecer análises mais rápidas e fáceis de usar que o *Hadoop* MapReduce.

Antes da Apache Software Foundation tomar posse do *Spark*, este estava sob o controle da Universidade da Califórnia, o laboratório AMP de Berkeley. Além disso, o *Spark* pode ser implantado de várias maneiras, como no *Machine Learning*, no *streaming* de dados e no processamento de gráficos.

O *Spark* usa as bibliotecas clientes do *Hadoop* para HDFS e YARN, utiliza muitos outros componentes, como HDFS. O Apache *Spark* não fornece qualquer armazenamento (como HDFS) ou qualquer meio de gerenciamento de recursos, ele é apenas uma estrutura unificada para processar grande quantidade de dados perto do tempo real. O *Spark* é independente do *Hadoop*, pois possui seu próprio sistema de gerenciamento de *cluster*, usando o *Hadoop* apenas para fins de armazenamento. Muitos autores colocam o *Spark* no esquema de ecossistemas do *Hadoop*. Existe uma intimidade muito grande entre eles, pois, em muitos projetos, as empresas optam por trabalhar com ambos, conjuntamente.

Seguem alguns recursos importantes que fizeram o Apache *Spark* ser uma das tecnologias preferidas do mundo *Big Data*:

» Processamento rápido

O *Big Data* é caracterizado por volume, variedade, velocidade e veracidade, que precisa ser processado em uma velocidade mais alta. O *Spark* contém RDD (*Resilient Distributed Dataset*), que economiza tempo nas operações de leitura e gravação, permitindo que seja executado quase dez a cem vezes mais rápido que o *Hadoop*.

» Flexibilidade

O Apache *Spark* suporta vários idiomas e permite que os desenvolvedores escrevam aplicativos em Java, Scala, R ou Python.

» Computação na memória

O *Spark* armazena os dados na RAM dos servidores, o que permite acesso rápido e, por sua vez, acelera a velocidade da análise.

» Processamento em tempo real

Diferentemente do MapReduce, que processa apenas dados armazenados, o *Spark* é capaz de processar dados de *streaming* em tempo real e, portanto, é capaz de produzir resultados instantâneos.

» Melhores análises

O Apache *Spark* consiste em um rico conjunto de consultas SQL, algoritmos de aprendizado de máquina, análises complexas etc. Com todas essas funcionalidades, a análise pode ser realizada de uma maneira melhor que a do MapReduce.

Segurança

A segurança no *Spark* está desativada por padrão, ou seja, antes da instalação, é muito importante incluir o funcionamento do *Spark* na política de segurança da organização. Algumas recomendações de configuração do *Spark*:

» Autenticação

O *Spark* atualmente suporta autenticação para canais RPC usando um segredo compartilhado. A autenticação pode ser ativada definindo o parâmetro de configuração `spark.authenticate`.

» YARN *Spark*

Em implementações locais do YARN, o *Spark* manipulará automaticamente a geração e distribuição do segredo compartilhado, cada aplicativo usará um segredo compartilhado exclusivo e, no caso do YARN, esse recurso conta com a criptografia YARN RPC ativada para a distribuição de segredos para ser segura.

» Gerenciadores de terceiros

Para outros gerenciadores de recursos, o `spark.authenticate.secret` deve ser configurado em cada um dos nós. Esse segredo será compartilhado por todos os *daemons* e aplicativos, portanto, essa configuração de implantação não é tão segura quanto a anterior, especialmente ao considerar *clusters* com vários inquilinos, em que um usuário com o segredo pode efetivamente representar qualquer outro usuário.

» Criptografia

O *Spark* suporta criptografia baseada em AES para conexões RPC. Para que a criptografia seja ativada, a autenticação RPC também deve estar ativada e configurada corretamente. A criptografia AES usa a biblioteca *Apache Commons Crypto*, e o sistema de configuração do *Spark* permite acesso à configuração dessa biblioteca para usuários avançados.

Existem várias formas de configurar a segurança do *Spark*. Para isso, a documentação dos parâmetros é disponibilizada na *Spark* Security:

» Criptografia de armazenamento local.

» UI da Web – Autenticação e autorização.

» ACLs do Servidor do Histórico *Spark*.

» Configuração SSL.

» Cabeçalhos de segurança HTTP.

» Portas para segurança de rede.

» Kerberos.

» Registro de eventos.

Veja como o *framework* Apache *Spark* está organizado em três camadas principais:

Tabela 9. Composição do *Spark*.

Spark SQL	*Spark* *Streaming*	Machine Learning Library	*GraphX* processing
Spark Core			
Standalone Scheduler		YARN	Mesos

Fonte: O autor.

Algumas vantagens do *Spark* em relação ao *Hadoop* podem ser analisadas através da documentação da API de programação do *Spark*.

» Processamento na memória

Foi visto no capítulo anterior que o MapReduce possui uma desvantagem quando se usa somente *Hadoop*. A comparação aqui é no processamento dos dados, que, por processar os dados em memória e não haver movimentação de dados, não se perde tempo e obtém um ganho 100 vezes mais rápido que o modo de memória do *Hadoop* e 10 vezes mais rápido que o modo de disco do *Hadoop*.

» Processamento de fluxo

O Apache *Spark* suporta o processamento de fluxo, que envolve entrada e saída contínuas de dados; por esse motivo, também é chamado de processamento em tempo real.

» Menor latência

O RDD (*Resilient Distributed Dataset*) gerencia o processamento distribuído de dados e a transformação desses dados. Cada conjunto de dados em um RDD é particionado em partes lógicas, que podem ser computadas em diferentes nós de um *cluster*.

» Avaliação preguiçosa

O Apache *Spark* começa a avaliar apenas quando é absolutamente necessário. Isso desempenha um papel importante na contribuição para sua velocidade.

» Menos linhas de código

Embora o *Spark* seja escrito no Scala e no Java, a implementação está no Scala, portanto, o número de linhas é relativamente menor no *Spark* quando comparado ao *Hadoop*.

Deixando as diferenças de lado, muitas empresas que implementam projetos de *Big Data* consideram a utilização de ambas as tecnologias para resolverem diversos desafios em relação ao processamento e extração de valores dos dados.

CAPÍTULO 2
Componentes do *Spark*

O *Spark* é composto por vários componentes. Assim como o *Hadoop*, as necessidades foram surgindo e ferramentas foram sendo desenvolvidas para integrá-lo.

Spark Core

O *Spark Core* é o componente básico do *Spark*, que inclui todos os componentes para agendamento de tarefas, executando várias operações de memória, tolerância a falhas e muito mais. O *Spark Core* também abriga a API, que consiste em RDD. Além disso, o *Spark Core* fornece APIs para criar e manipular dados no RDD. Essas funcionalidades fornecidas pelo Apache *Spark* são construídas na parte superior do *Spark Core*, que oferece velocidade, fornecendo capacidade de computação na memória. Dessa forma, o *Spark Core* é a base do processamento paralelo e distribuído de grandes conjuntos de dados.

Seus principais recursos são:

- É responsável por funcionalidades essenciais de E / S.
- Significativo na programação e observação do papel do *cluster Spark*.
- Controle de tarefas.
- Recuperação de falhas.
- Suas lógicas em memória, resolvendo o problema do MapReduce.

No *Spark Core*, é possível utilizar uma coleção de abstrações especial chamada RDD (*Resilient Distributed Dataset*). O *Spark* RDD manipula os dados que estão particionados em memória dos *clusters* e os mantém no conjunto de memórias de uma única unidade. Com isso, ele pode executar duas operações: a transformação, que produzirá um novo DataSet a partir de outros existentes e/ou uma ação, que ao invés de utilizar um DataSet existente, trabalha com um DataSet de dados reais. No capítulo 3 da Unidade IV serão detalhadas as demais funcionalidades do RDD.

Spark SQL

O componente *Spark* SQL é uma estrutura distribuída para processamento de dados estruturados. O *Spark* SQL permite consultar dados via SQL, bem como através do formulário de SQL do Apache Hive, chamado *Hive Query Language* (HQL). Ele também suporta dados de várias fontes, como tabelas de análise, arquivos de log, JSON etc. O *Spark* SQL permite que os programadores combinem consultas SQL com alterações ou manipulações programáveis suportadas pelo RDD em Python, Java, Scala e R; com isso, é possível obter mais informações das estruturas de dados, podendo trabalhar com dados semi e não-estruturados.

Seus recursos incluem:

- Otimização baseada em custos.
- Tolerância a falhas durante uma consulta.
- Compatibilidade total com os dados existentes do Hive. (https://cwiki.apache.org/confluence/display/Hive/LanguageManual).
- Acesso a diversas fonte de dados: Hive, Avro, Parquet, ORC, JSON e JDBC.
- Possibilidade para transportar dados estruturados dentro dos programas *Spark*, usando SQL ou uma API familiar do Data Frame.

Spark Streaming

O *Spark Streaming* é um complemento para o *Spark Core* que processa os dados gerados por várias fontes e são processados instantaneamente, permitindo processamento escalável, de alta taxa de transferência e tolerante a falhas de fluxos de dados ativos.

O *Spark Streaming* pode acessar dados de fontes como *Kafka, Flume, Kinesis* ou soquete TCP. Ele utiliza a técnica de microlotes, que permite que um processo ou tarefa trate um fluxo como uma sequência de pequenos lotes de dados.

Resumindo, ele agrupa os dados em tempo real em pequenos lotes, entrega aos algoritmos separadamente para processamento, protege de falhas e entregas os dados para serem expostos.

As fases do *Spark Streaming* são:

» Gathering

› **Fontes de dados b**ásicas: disponíveis na API do *Core*.

› **Fontes avançadas:** soquetes *Kafka, Flume, Kinesis* ou TCP.

» Processing

Utiliza as opções de transformação e ação.

» Data Storage

Depois de processados, os dados são enviados a diversas saídas, como banco de dados e/ou painéis ativos.

GraphX

GraphX é a biblioteca do Apache *Spark* para aprimorar gráficos e permitir a computação paralela a gráficos. O Apache *Spark* inclui vários algoritmos de gráficos que ajudam os usuários a simplificar a análise de gráficos. É o mecanismo de análise de gráficos de rede e o armazenamento de dados. Também é possível agrupar, classificar, atravessar, pesquisar e localizar caminhos nos gráficos. Além disso, o *GraphX* amplia o *Spark* RDD, trazendo à luz uma nova abstração do *Graph*: um *multigraph* direcionado com propriedades anexadas a cada vértice e aresta.

MLlib

O Apache *Spark* cria uma biblioteca que contém serviços comuns de *Machine Learning* (ML) chamados *MLlib*. Ele fornece vários tipos de algoritmos de ML, incluindo regressão, *clustering* e classificação, que podem executar várias operações nos dados para obter informações significativas sobre ele, é escalável, proporciona algoritmos de alta qualidade e alta velocidade.

SparkR

É um pacote R que fornece um *front-end* leve para usar o Apache *Spark* da plataforma R. O *SparkR* fornece uma implementação de *DataFrame* distribuído

que suporta operações como seleção, filtragem, agregação etc., iguais aos quadros de dados R, *dplyr*, mas em grandes conjuntos de dados. O *SparkR* também faz uma boa integração e suporta aprendizado de máquina distribuído usando o *MLlib*.

SparkDataFrame

Um *SparkDataFrame* é uma coleção distribuída de dados organizados em colunas nomeadas. É conceitualmente equivalente a uma tabela em um banco de dados relacional ou um quadro de dados em R, mas com otimizações encapsuladas mais avançadas. Os *SparkDataFrames* podem ser construídos a partir de uma ampla variedade de fontes, como: arquivos de dados estruturados, tabelas no Hive, bancos de dados externos ou quadros de dados R existentes.

Instalação do *Spark*

Para que o *Spark* funcione, não é preciso de uma instalação propriamente dita. Apenas é preciso executar seus *scripts* de execução. O mais demorado é a preparação do ambiente e sua configuração.

O sistema operacional mais indicado para sua utilização é o Linux, portanto, o exemplo de execução será feito nesse S.O. Sua simulação poderá ser feita em uma máquina virtual. Nesse caso, foi utilizada a máquina virtual da Orcale, a VirtualBox.

A distribuição Linux utilizada foi a Lubuntu. Ela é uma distribuição leve que não utilizará muito recurso da VM.

Logo após o carregamento do Linux, faça o login e instale o Java, pois é pré-requisito para funcionamento do *Spark*.

O primeiro comando é para atualizar o repositório de seu S.O:

```
$sudo apt update
```

Em seguida, é preciso verificar se existe alguma versão do Java instalado. Por padrão, o Lubuntu não contém o Java instalado. Para verificar, digite o comando:

```
$java –version
```

Caso não possua, aparecerá a seguinte mensagem:

```
Command 'java' not found, but can be installed with:
apt install default-jre
apt install openjdk-11-jre-headless
apt install openjdk-8-jre-headless
```

Se for preciso efetuar a instalação, basta executar o seguinte comando:

```
$sudo apt install default-jre
```

E verificar se instalou corretamente:

```
java -version
openjdk version "10.0.2" 2018-07-17
OpenJDK Runtime Environment (build 10.0.2+13-Ubuntu-
1ubuntu0.18.04.4)
OpenJDK 64-Bit Server VM (build 10.0.2+13-Ubuntu-
1ubuntu0.18.04.4, mixed mode)
```

É preciso também instalar as bibliotecas de desenvolvimento do Java JDK. Para isso, execute o seguinte comando:

```
$sudo apt install default-jdk
```

E verificar se a instalação foi efetuada com sucesso:

```
$javac -version
```

Agora, é preciso baixar os pacotes do *Spark*. Para isso, acesse o link http://spark.apache.org/downloads.html, escolha o link da opção 3 ou baixe desse link: https://www.apache.org/dyn/closer.lua/spark/spark-3.0.0-preview2/spark-3.0.0-preview2-bin-hadoop2.7.tgz. O último acesso do link foi dia 28/12/2019, ou seja, as versões serão atualizadas sempre e é recomendável escolher as versões mais atualizadas. Ou, ainda, é possível baixar através do utilitário *wget*. O *wget* é um utilitário de linha de comando para baixar arquivos da Internet. Ele suporta o download de vários arquivos, o download em segundo plano, a retomada de downloads, limitando a largura de banda usada para downloads e visualização de cabeçalhos.

```
$wget  "https://www.apache.org/dyn/closer.lua/spark/
spark-3.0.0-preview2/spark-3.0.0-preview2-bin-
hadoop2.7.tgz"
```

Logo que o download terminar, é preciso fazer a descompactação do arquivo através do *tar*. O *tar* é um utilitário de linha de comando para compactar arquivos e/ou agrupá-los. Normalmente é chamado de tarball, ou tar, gzip e bzip no Linux. O algoritmo usado para compactar um arquivo *.tar.gz* e *.tar.bz2* são algoritmos gzip ou bzip, respectivamente. A maioria das distribuições já vem com a instalação desses utilitários por padrão.

Para isso, execute o comando:

```
$tar –zfx spark-3.0.0-preview2-bin-hadoop2.7.tgz
```

Esse comando descompacta o pacote "tgz" e cria um diretório com o mesmo nome. Depois de compactado, acesse o diretório para efetuar alguns comandos de checagem:

```
$cd spark-3.0.0-preview2-bin-hadoop2.7 && ls -la
```

Deverá aparecer a seguinte lista de diretórios:

```
drwxr-xr-x 13 user user  4096 dez 17 01:48 .
drwxr-xr-x 17 user user  4096 jan  2 17:53 ..
drwxr-xr-x  2 user user  4096 dez 17 01:48 bin
drwxr-xr-x  2 user user  4096 dez 17 01:48 conf
drwxr-xr-x  5 user user  4096 dez 17 01:48 data
drwxr-xr-x  4 user user  4096 dez 17 01:48 examples
drwxr-xr-x  2 user user 16384 dez 17 01:48 jars
drwxr-xr-x  4 user user  4096 dez 17 01:48 kubernetes
-rw-r--r--  1 user user 23311 dez 17 01:48 LICENSE
drwxr-xr-x  2 user user  4096 dez 17 01:48 licenses
-rw-r--r--  1 user user 57677 dez 17 01:48 NOTICE
drwxr-xr-x  9 user user  4096 dez 17 01:48 python
drwxr-xr-x  3 user user  4096 dez 17 01:48 R
-rw-r--r--  1 user user  4666 dez 17 01:48 README.md
-rw-r--r--  1 user user   192 dez 17 01:48 RELEASE
drwxr-xr-x  2 user user  4096 dez 17 01:48 sbin
drwxr-xr-x  2 user user  4096 dez 17 01:48 yarn
```

Para verificar se tudo ocorreu bem, execute o seguinte comando:

```
$./bin/pyspark
```

O resultado deverá ser:

```
Welcome to SPARK version 3.0.0-preview2
Using Python version 2.7.17 (default, Nov  7 2019
10:07:09)
SparkSession available as 'spark'.
>>>
```

Se tudo funcionou bem, o *Spark* fornecerá um *prompt* do tipo REPL, ou seja, um REPL irá receber entrada (leitura), executar esses comandos (avaliar) e imprimir os resultados, tudo em um *loop*. Isso é possível porque os desenvolvedores criaram facilidade no comando *pyspark*, interagindo com a linguagem Python. Esse terminal é um terminal interativo que receberá comandos Python e enviará ao *Spark Core*. Isso funciona para as demais linguagens de programação habilitadas para funcionar na API do *Spark*.

E o *Spark* estará pronto para receber comandos. Para sair, execute:

```
>>> quit()
```

ou

```
>>> control + D
```

Que é um comando Python.

CAPÍTULO 3
RDDs

RDD (*Resilient Distributed Datasets* – DataSets Distribuídos e Resilientes) é a estrutura de dados fundamental do Apache *Spark*, que é uma coleção imutável de objetos que computa nos diferentes nós do *cluster*. Todo e qualquer conjunto de dados no *Spark* RDD é particionado logicamente em muitos servidores, para que possam ser computados em diferentes nós do *cluster*.

Segundo RDD Programming Guide (2019), um RDD pode ser:

> Os RDDs são uma coleção de objetos semelhantes às coleções no Scala, com a diferença de que o RDD é calculado em várias JVMs espalhadas por vários servidores físicos, também chamados de nós em um cluster, enquanto uma coleção Scala vive em uma única JVM. Eles fornecem abstração de dados do particionamento e distribuição dos dados projetados para executar cálculos em paralelo em vários nós, enquanto fazem transformações no RDD na maioria das vezes, não precisamos nos preocupar com o paralelismo.

Alguns recursos do *Spark* RDD:

- Computação em memória RAM.
- Tolerância a falhas.
- Avaliação preguiçosa (*lazy*).
- Imutabilidade.
- Particionamento.
- Granulação grossa (*coarse-grained*).

Os RDDs são imutáveis (somente leitura) por natureza, isso quer dizer que não é possível alterar um RDD original, porém pode criar novos RDDs executando operações de transformações, em um RDD existente. Resilientes por serem tolerantes a falhas e capazes de recompilar partições ausentes ou danificadas devido a falhas nos nós.

Outra informação importante está no RDD Programming Guide (2019):

> Uma das abstrações no *Spark* são *variáveis compartilhadas* que podem ser usadas em operações paralelas. Por padrão, quando o *Spark* executa uma função em paralelo como um conjunto de tarefas em diferentes nós, ele envia uma cópia de cada variável usada na função para cada tarefa. Às vezes, uma variável precisa ser compartilhada entre tarefas ou entre tarefas e o programa do driver. O *Spark* suporta dois tipos de variáveis compartilhadas: variáveis de *broadcast*, que podem ser usadas para armazenar em cache um valor na memória em todos os nós, e *acumuladores*, que são variáveis que são apenas "adicionadas" a, como contadores e somas. (SPARK, 2019)

Essa explicação mostra a dimensão das funcionalidades do *Spark* e o RDD quando o assunto é programação distribuída. Uma das grandes vantagens do RDD é o seu processamento em memória, e isso fez com que vários desenvolvedores migrassem suas aplicações a essa nova arquitetura.

Antes do RDD (e ainda hoje existem várias aplicações sendo desenvolvidas assim), o recurso que as equipes de desenvolvimento tinham para melhorar o desempenho das aplicações era a programação **distribuída em memória compartilhada (DSM)**. Esse tipo de programação depende exclusivamente das habilidades das equipes de Dev e Ops. Ela dificulta a implementação de maneira eficiente e tolerante a falhas nos *clusters*. Em outra tentativa de melhorar essa performance, foi criado o HDFS do *Hadoop*, que tornaria o cálculo das tarefas mais lento, pois envolve muitas operações, replicações e serializações de E/S no processo.

Entretanto, a grande dificuldade do RDD era definir um mecanismo que provesse tolerância a falhas com eficiência. Para resolver esse problema, foi utilizada a granulação grossa sob a granulação fina do HDFS. A granulação grossa transforma um conjunto específico de dados e não somente um registro, enquanto a granulação fina transforma um único registro dentro de um conjunto de dados.

Outras vantagens que a granulação grossa possui sob a granulação fina está na implementação em mecanismo relacionadas à recuperação de falhas. No RDD, os dados perdidos poderão ser recuperados graças à imutabilidade. Nas tecnologias DSM, para poder fazer a recuperação de dados é necessário retornarem ao ponto de verificação mais recente, ou seja, se o ponto foi verificado 10 horas antes da falha, teremos problemas em recuperar os dados que foram criados depois.

Outro problema em relação às tecnologias DSM é o tratamento da memória RAM. Caso não haja mais espaço livre na memória, o RDD desloca os dados inativos para o disco. Já no DSM, a troca entre a memória RAM e o disco é constante.

Os RDDs no *Spark* possuem duas operações básicas: a de transformação e ação.

- **Transformações** são métodos que aceitam RDDs existentes como entrada e produzem um ou mais RDDs. Como os dados no RDD não são alterados (imutáveis), serão produzidos novos RDDs, através de operações de transformação, como *map, reduceByKey, reduceByKey, union, intersection* etc.

- **Ações**, por sua vez, são as funções que retornarão os resultados das transformações de RDD. Elas fornecerão valores e não um novo RDD. Algumas delas são: *first, take, collect, reduce, count* etc. Quando se utiliza transformações, existe a possibilidade de criar novos RDDs a partir de um existente, ou seja, a ação é um complemento da transformação, elas são operações que não fornecem valores RDD.

Vantagens

- Processamento na memória.
- Imutabilidade.
- Tolerância a falhas.
- Evolução preguiçosa.
- Particionamento.
- Paralelismo.

Desvantagens

- Não possui mecanismos de otimização.
- Dificuldade com dados estruturados (não possuem inferências).
- Desempenho limitado devido ao Garbage Collector da JVM.
- Limitação em armazenamento.

Programação

Os RDDs são criados iniciando com um arquivo no sistema de arquivos suportado pelo *Hadoop*. Os usuários também podem solicitar ao *Spark* que mantenha um RDD na memória, permitindo que ele seja reutilizado com eficiência em operações paralelas. Os RDDs podem conter qualquer tipo de objetos Python, Java ou Scala, incluindo classes definidas pelo usuário. Vamos utilizar os exemplos da documentação do RDD e os dados que foram utilizados.

Primeiramente, se o *Spark* estiver rodando, dê um *control + D* para pará-lo. Isso se faz necessário porque será preciso inicializá-lo de outra forma:

```
$ ./bin/pyspark --master local[4]
```

Em seguida, com o *Spark* já inicializado, é preciso modificar a variável *SparkSession* para que seja carregada com alguns parâmetros:

```
>>> spark = SparkSession.builder\
    .master("local")\
    .appName("Aula")\
    .config("spark.executor.memory","1gb")\
    .getOrCreate()
```

Agora, é preciso criar um objeto do tipo *SparkContext*:

```
>>> sc = spark.sparkContext
```

Para carregar os dados, utilizaremos o arquivo no formato "csv" e não do tipo *DataFrame*. Para isso, digite o seguinte comando:

```
>>> datafile = sc.textFile("/home/user/uni4cap4.utf8.
csv")
```

Depois do arquivo carregado, já podemos utilizar as operações de transformação e ação.

```
>>> datafile.count()
```

CAPÍTULO 4
DataSets, DataFrames e *Spark* SQL

Segundo o próprio guia de programação do *Spark*:

> ***Spark SQL*** é um módulo *Spark* para processamento de dados estruturados que, diferentemente da API básica do RDD do *Spark*, as interfaces fornecidas pelo *Spark* SQL fornecem ao *Spark* mais informações sobre a estrutura dos dados e da computação que está sendo executada. (SPARK, 2019)

Existem várias maneiras de interagir com o *Spark* SQL, incluindo o SQL e a API do *DataFrame*, ou seja, os desenvolvedores podem alternar facilmente entre diferentes APIs.

Um DataSet é uma coleção distribuída de dados, é uma nova interface adicionada no *Spark* que fornece os benefícios dos RDDs com os benefícios do mecanismo de execução otimizado do *Spark* SQL. A API do DataSet está disponível no Scala e Java. O Python e o R não têm suporte para a API, porém, devido à natureza dinâmica dessas linguagens de programação, muitos dos benefícios da API do DataSet já estão disponíveis.

Um *DataFrame* é um conjunto de dados organizado em colunas nomeadas com as mesmas características do *Spark DataFrame*, mencionadas no Capítulo 2, Componentes.

Criando um *DataFrame* no *Spark*

O *DataFrame* que o *Spark* utiliza é semelhante aos *DataFrames* de todas as linguagens compatíveis com a API, portanto, seja qual for a linguagem da API que estiver sendo utilizada, o *Spark* irá entender como *DataFrame*.

Basicamente, um *DataFrame* em todas as linguagens da API é uma estrutura de dados bidimensional, ou seja, os dados são alinhados de forma tabular em linhas e colunas. O *DataFrame* consiste em três componentes principais: os dados, as linhas e as colunas. No mundo real, um *DataFrame* será criado carregando os conjuntos de dados do armazenamento existente, que podem ser Banco de Dados SQL, arquivo CSV e arquivo do Excel, arquivo JSON etc. O *DataFrame* também pode ser criado a partir das listas, do dicionário e de uma lista de

dicionário etc. Vamos ver na prática e todos os exemplos serão adaptados da documentação Python Programming Guide (2019).

Primeiramente, é preciso iniciar o *Spark* e aguardar o terminal interativo. Em seguida, digite o seguinte código no terminal:

```
>> spark
<pyspark.sql.session.SparkSession object at
0x7f5e1e74e510>
```

Esse comando verifica a variável de sessão que é iniciada com o *Spark*. Ela é o ponto de entrada para a programação do *Spark* com o DataSet e a API *DataFrame*. Veja as principais classes do *Spark* SQL e *DataFrames*:

- ***pyspark.sql.SparkSession:*** principal ponto de entrada *DataFrame* e funcionalidade do SQL.
- ***pyspark.sql.DataFrame:*** uma coleção distribuída de dados agrupados em colunas nomeadas.
- ***pyspark.sql.Column:*** uma expressão de coluna em *DataFrame*.
- ***pyspark.sql.Row:*** uma linha de dados em *DataFrame*.
- ***pyspark.sql.*GroupedData*:*** métodos de agregação, retornados por *DataFrame*.groupBy().
- ***pyspark.sql.DataFrameNaFunctions:*** métodos para manipular dados ausentes (valores nulos).
- ***pyspark.sql.DataFrameStatFunctions:*** métodos para funcionalidade estatística.
- ***pyspark.sql.functions:*** lista de funções internas disponíveis para *DataFrame*.
- ***pyspark.sql.types:*** lista de tipos de dados disponíveis.
- ***pyspark.sql.Window:*** para trabalhar com funções da janela.

Um *SparkSession* pode ser usado para criar *DataFrame*, registrar *DataFrame* como tabelas, executar SQL sobre tabelas, armazenar em cachê tabelas e ler arquivos.

É possível criar um *DataFrame* manualmente, como um teste ou exemplo, ou utilizar a maneira mais usual devido à quantidade de dados.

```
>>>
dta = [("jorge",45),("maria",50),("marta",33),("pedro",34)]
>>> df=spark.createDataFrame(dta,["Nomes","Idades"])
>>> df.show()
```

Como exemplo, foi utilizado o arquivo baixado do Portal da Transparência. Nele, é possível escolher o período (2019) e baixar o arquivo. Foram removidas algumas colunas e alteradas seus nomes, ficando assim: PaisOrigem, UFOrigem, CidadeOrigem, PaisDestino, UFDestino, CidadeDestino, Valor.

Em seguida, o arquivo foi modificado para o padrão de acentuação UTF-8 através do comando:

```
$ iconv -f ISO-8859-1 -t UTF-8 uni4cap4.csv > uni4cap4.utf8.
csv
```

O arquivo também estará disponibilizado na biblioteca da disciplina, chamado "uni4cap4.csv".

Para iniciar os exemplos, faça o download e salve no diretório "/home" e, em seguida, digite o seguinte comando:

```
>> df=spark.read.load("/home/user/uni4cap4.utf8.csv",
                      header="true",
                      format="csv",
                      sep=";",
                      inferSchema="true")
>>> df.show()
```

O primeiro comando carregou o arquivo para dentro da variável "df", e o segundo mostrará o cabeçalho e os primeiro registros. Nesse momento, a variável "df" conterá todos os métodos da classe *DataFrame*, ou seja, não é uma simples variável, ela é um objeto.

Sempre que se inicia um trabalho de Ciência de Dados, é obrigatório conhecer a estrutura de seus dados. Com o *Spark*, isso é possível através de vários comandos:

```
>>> df.printSchema()
    root
     |-- PaisOrigem: string (nullable = true)
     |-- UFOrigem: string (nullable = true)
```

```
|-- CidadeOrigem: string (nullable = true)
|-- PaisDestino: string (nullable = true)
|-- UFDestino: string (nullable = true)
|-- CidadeDestino: string (nullable = true)
|-- Valor: string (nullable = true)
```

Em certas situações, é preciso escolher apenas as colunas necessárias. Para isso, basta utilizar o método "*select*":

```
>>> df.select("PaisOrigem","PaisDestino","Valor").show()
```

Outra função essencial para se trabalhar com dados é o filtro.

```
>>>  df.select("PaisOrigem","PaisDestino","Valor").filter
(df["PaisDestino"]=="Itália").show()
```

Os filtros são muito importantes para que recortemos massas de dados para utilização específica. Entretanto, existe uma maneira mais fácil de efetuar consultas, pelo menos para a maioria dos desenvolvedores tradicionais: através do SQL. A função *sql* da *SparkSession* permite que os aplicativos executem consultas SQL programaticamente e retorna o resultado como um *DataFrame*.

Para isso, é preciso criar uma visão temporária que receberá as consultas. As visões temporárias no *Spark* SQL têm escopo de sessão e desaparecem se a sessão que o cria terminar.

```
>>> df.createOrReplaceTempView("viagens")
>>> sqlDf=spark.sql("select * from viagens")
>>> sqlDf.show()
>>> sqlDf.count()
```

Se for preciso ter uma exibição temporária compartilhada entre todas as sessões e manter-se ativo até o aplicativo *Spark* terminar, é possível criar uma exibição temporária global.

```
>>> df.createGlobalTempView("viagens1")
```

A visão temporária global está vinculada a um banco de dados preservado pelo sistema *global_temp*, e devemos usar o nome qualificado para consultá-lo, por exemplo:

```
>>> sqlDf=spark.sql("select * from global_temp.viagens1")
>>> sqlDf.show()
>>> sqlDf.count()
```

Perceba que criamos um novo *DataFrame* com as informações retornadas da consulta SQL. Portanto, toda vez que for preciso utilizar filtros, é preciso colocar em um novo objeto:

```
>>> sqlDfBA=spark.sql("select * from viagens where
UFOrigem ='Bahia'")
>>> sqlDfBA.show()
>>> sqlDf.count()
```

Os *DataFrames* também pode ser salvos como tabelas persistentes no *metastore* do Hive usando o método *saveAsTable*. Diferentemente do comando *createOrReplaceTempView*, *saveAsTable* materializará o conteúdo do *DataFrame* e criará um ponteiro para os dados no *metastore* do Hive. As tabelas persistentes ainda existirão mesmo após o reinício do seu programa *Spark*, desde que você mantenha sua conexão com o mesmo *metastore*.

Para fonte de dados baseada em arquivo, como texto, parquet, JSON etc., é possível especificar um caminho de tabela personalizado através da opção *df.write.option("path", "/some/path").saveAsTable("t")*.

Quando a tabela é descartada, o caminho da tabela personalizada não será removido e os dados da tabela ainda estarão lá. Se nenhum caminho de tabela personalizado for especificado, o *Spark* gravará dados em um caminho de tabela padrão no diretório do armazém. Quando a tabela é descartada, o caminho da tabela padrão também será removido.

No caso do exemplo, pode digitar:

```
>>> sqlDfBA.write.option("/home/user/").saveAsTable("fileBA.csv")
```

Referências

ALVES, M. J. P. **Construindo supercomputadores com Linux**. 1. ed. Rio de Janeiro: Brasport, 2002.

AMAZON. **NoSQL**. Disponível em: https://aws.amazon.com/pt/nosql/. Acesso em: 12 dez. 2019.

APACHE. **HQL**. Disponível em: https://cwiki.apache.org/confluence/display/Hive/LanguageManual. Acesso em 12 dez. 2019.

APACHE. **PIG**. Disponível em: https://cwiki.apache.org/confluence/display/PIG/Index. Acesso em: 12 dez. 2019.

APACHE. HBase. Disponível em: https://hbase.apache.org/. Acesso em: 12 dez. 2019.

APACHE. **Mahout**. Disponível em: https://mahout.apache.org/. Acesso em: 12 dez. 2019.

APACHE. **ZooKeeper**. Disponível em: https://zookeeper.apache.org/. Acesso em: 12 dez. 2019.

APACHE. **Oozie**. Disponível em: https://oozie.apache.org/. Acesso em: 12 dez. 2019.

APACHE. **Sqoop**. Disponível em: https://sqoop.apache.org/. Acesso em: 12 dez. 2019.

APACHE. **Flume**. Disponível em: https://flume.apache.org/. Acesso em: 12 dez. 2019.

APACHE. **Ambari**. Disponível em: https://ambari.apache.org/. Acesso em: 12 dez. 2019.

APACHE. **Drill**. Disponível em: https://drill.apache.org/. Acesso em: 12 dez. 2019.

APACHE. **Lucene**. Disponível em: https://lucene.apache.org/solr/. Acesso em: 12 dez. 2019.

APACHE. **Apache Hadoop YARN**. Disponível em: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html. Acesso em: 12 dez. 2019.

APACHE. **Hadoop MapReduce**. Disponível em: http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. Acesso em: 12 dez. 2019.

BEAL, A. **Gestão Estratégica da Informação**. Como transformar a informação e a Tecnologia em Fatores de Crescimento e de Alto Desempenho nas Organizações. São Paulo: Editora Atlas, 2004.

BRASIL. Portal da Transparência. **Arquivo de Gastos com Viagens**. http://transparencia.gov.br/download-de-dados/viagens. Acesso em: 12 dez. 2019.

BROWN, R. G. **What makes a Cluster Beowulf?** Duke University Physics Department, 2006. Disponível em: www.beowulf.org/overview/howto.html. Acesso em: 12 dez. 2019.

CHEN, Peter. **The entity-relationship model** - toward a unified view of data. Massachusetts Institute of Technology. Vol. 1, Março de 1976.

DATA FLAIR. **Top Advantages and Disadvantages of *Hadoop* 3**. Disponível em: https://data-flair.training/blogs/advantages-and-disadvantages-of-hadoop. Acesso em: 12 dez. 2019.

DAVENPORT, Thomas H.; PRUSAK, L. **Working Knowledge**: How Organizations Manage What They Know, 2005. Disponível em: https://www.researchgate.net/publication/229099904. Acesso em: 01 dez. 2019.

DUTTA, P. **Top 20 Best Big Data Applications & Examples in Today's World.** s/d. Disponível em: https://www.ubuntupit.com/best-big-data-applications-in-todays-world. Acesso em: 12 dez. 2019.

GEIST, A. *et al*. **PVM**: *Parallel Virtual Machine*: a user's Guide and Tutorial for Networked Parallel Computing. Cambridge: MIT Press, 1994. Disponível em: www.netlib.org/pvm3/book/pvm-book.html. Acesso em: 12 dez. 2019.

IDC IVIEW - **The Digital Universe in 2020:** *Big Data*, Bigger Digital Shadows, and Biggest Growth in the Far East. 2012. Disponível em: https://www.emc.com/leadership/digital-universe/2012iview/big-data-2020.htm. Acesso em: 12 dez. 2019.

IBM. **Communication Programming Concepts.** Disponível em: https://www.ibm.com/support/knowledgecenter/beta/pt/ssw_aix_71/commprogramming/progcomc_kickoff.html. Acesso em: 12 dez. 2019.

JENSEN Christian S. **ACM Transactions on Database Systems**. NY: Association for Computing Machinery, 1976.

MARR, B. **WHAT IS BIG DATA**? Disponível em: https://www.bernardmarr.com/default.asp?contentID=766. Acesso em: 12 dez. 2019.

MONGODB. Disponível em: https://www.mongodb.com/. Acesso em: 12 dez. 2019.

MORIMOTO, C. E. Brincando de *Cluster*. **Guia do Hardware**, 2003. Disponível em: www.guiadohardware.net/artigos/cluster/. Acesso: 12 dez. 2019.

SPARK. **Python Programming Guide**. Disponível em: https://spark.apache.org/docs/0.9.1/python-programming-guide.html. Acesso em: 12 dez. 2019.

SPARK. **RDD Programming Guide**. Disponível em: https://spark.apache.org/docs/latest/rdd-programming-guide.html. Acesso em: 12 dez. 2019.

SPARK. **API de programação do *Spark***: https://spark.apache.org/docs/latest/streaming-programming-guide.html. Aceso em 12 dez. 2019.

SPARK. **Spark Security: Criptografia e armazenamento local**. Disponível em: https://spark.apache.org/docs/latest/security.html. Acesso em 12 dez. 2019.

TANENBAUM, Andrew S.; STEEN, Maarten van. **Sistemas Distribuídos**: princípios e paradigmas. São Paulo: Pearson Prentice Hall, 2007.

TANENBAUM, A. S. **Organização estruturada de computadores**. 2. ed. Rio de Janeiro: Prentice/Hall do Brasil, 1992.

WEBER, R. F. **Arquitetura de computadores pessoais**. 2. ed. Porto Alegre: Sagra Luzaatto, 2003.

Sites

https://www.ibmbigdatahub.com/.

https://www.speicherguide.de/download/dokus/IDC-Digital-Universe-Studie-iView-11.12.pdf.

https://www.ibmbigdatahub.com/infographic/four-vs-big-data.

https://docs.mongodb.com/manual/core/databases-and-collections/.

https://docs.mongodb.com/manual/core/document/#bson-document-format.

https://www.ibm.com/support/knowledgecenter/ssw_aix_71/commprogramming/ch8_rpc.html.

https://cwiki.apache.org/confluence/display/Hive/LanguageManual.

https://twitter.com/cutting.

https://twitter.com/mikecafarella.

https://www.cloudera.com/products/open-source/apache-hadoop.html.

https://cwiki.apache.org/confluence/display/Hive/LanguageManual.

http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

https://research.google/research-areas/distributed-systems-and-parallel-computing/.

https://research.google/pubs/pub27898/.

https://hbase.apache.org/.

https://zookeeper.apache.org/.

https://sqoop.apache.org/.

https://flume.apache.org/.

https://drill.apache.org/.

https://lucene.apache.org/solr/.

https://www.bernardmarr.com/default.asp?contentID=766.

https://www.speicherguide.de/download/dokus/IDC-Digital-Universe-Studie-iView-11.12.pdf.

https://www.ibmbigdatahub.com/infographic/four-vs-big-data.

http://www.netlib.org/pvm3/book/node17.html.

https://www.mongodb.com/download-center?ct=false#community.

https://docs.mongodb.com/manual/crud/.

https://docs.mongodb.com/manual/reference/operator/query/.

https://www.cloudera.com/products/open-source/apache-hadoop.html.

https://aws.amazon.com/pt/emr/.

https://spark.apache.org/docs/latest/security.html.

https://spark.apache.org/docs/latest/streaming-programming-guide.html.

https://cwiki.apache.org/confluence/display/Hive/LanguageManual.

https://sparkbyexamples.com/spark-rdd-tutorial/.

https://docs.mongodb.com/manual/core/databases-and-collections/.

https://www.virtualbox.org/wiki/Downloads.

https://lubuntu.net/downloads.

http://spark.apache.org/downloads.html.

https://www.mongodb.com/nosql-explained.