



BIG DATA

AULA 1

Prof. Luis Henrique Alves Lourenço



CONVERSA INICIAL

Nesta aula, será apresentado a você um panorama sobre o conceito de Big Data. Estudaremos o contexto do qual surge esse conceito, os fundamentos que o definem e que são importantes ao tema. E abordaremos as etapas necessárias para extrair informações valiosas dos dados e apresentá-las.

TEMA 1 – A ERA DOS DADOS

Os avanços tecnológicos das últimas décadas nos trouxeram cada vez mais capacidade para medir e avaliar os eventos que acontecem a cada instante à nossa volta. A necessidade de gerar informações valiosas com base nos dados obtidos por tantos mecanismos de medição tem sido discutida e estudada a fim de combinar o uso de tais tecnologias para capturar, administrar e processar a quantidade cada vez maior de dados gerados das mais distintas formas.

Desde a popularização dos computadores e demais equipamentos digitais, vivemos em uma época de transição entre um mundo em que todos os dados eram gerados e armazenados em mídias analógicas para outro, em que os dados são digitais. Com essa revolução digital, surge a possibilidade de um volume muito significativo de dados a explorar. Enquanto nos anos 1990 havia poucos setores digitalizáveis, limitados a alguns segmentos da música e mídia, no início dos anos 2000 setores como o comércio eletrônico e o internet banking iniciaram sua transição para o digital, e hoje quase todos os aspectos da vida cotidiana passam por formatos digitais.

Dados da web, mídias sociais, transações das mais diversas naturezas (entretenimento, financeiro, telecomunicações), dados biométricos, relatórios, logs, documentos e muitos outros tipos de dados gerados a cada instante permitem a construção de aplicações que antes pareciam impossíveis devido ao alto custo e complexidade. O volume e a variedade dos dados gerados continuam a crescer, tornando sua análise cada vez mais complexa. No entanto, é cada vez mais necessário que tais análises ocorram, para que se possa produzir informações valiosas em tempo real.

Para responder à demanda pelas informações valiosas que podem ser obtidas com os dados, deve-se prestar especial atenção a certos fatores para processá-los e analisá-los, como: a relevância dos dados; a velocidade



necessária para processá-los; quão variados precisam ser; qual seu nível de atualização; entre diversos outros fatores que podem ser cruciais para as informações extraídas dos dados e que podem refletir a realidade e gerar o valor esperado.

Dessa forma, começamos a definir o conceito de *Big Data* não só como uma solução empacotada que pode ser colocada em prática adquirindo certa tecnologia com um fornecedor, mas como o conjunto de práticas e técnicas que envolvem o processamento de um volume de dados confiáveis e variados com a velocidade necessária à geração de valor.

1.1 O crescimento do volume de dados

Um dos aspectos mais influentes no crescimento dos dados foi a conexão entre os equipamentos digitais pela internet. Em 1995, quando a internet estava nos primórdios, estima-se que menos de 1% dos dados eram armazenados em formato digital (Marquesone, 2016). A conexão entre diversos dispositivos eletrônicos permitiu a criação de uma diversidade antes inimaginável de serviços que hoje são amplamente utilizados, como a compra de passagens on-line, definição de trajeto por auxílio de GPS, reuniões por videoconferência, serviços de financiamento coletivo, busca e candidatura de vagas de trabalho on-line, serviços de streaming de vídeo e áudio, redes e mídias sociais, jogos on-line, compras via comércio eletrônico, compras coletivas, internet banking, entre tantos outros.

Outro fator importantíssimo foi a adoção em grande escala de dispositivos móveis, que só foram intensamente popularizados devido à redução do custo de produção de equipamentos com poder de armazenamento adequado. Mesmo que houvesse a intenção de explorar os dados gerados, enquanto não houvesse poder de processamento ou capacidade de armazenamento suficiente a custos acessíveis, a maioria dos dados seria simplesmente descartada. Portanto, o aumento no poder de processamento, combinado com a redução de custo de armazenamento, contribuiu com o aumento do volume de dados.

As empresas puderam explorar o potencial contido em diferentes tipos de dados. Dados obtidos por diversos tipos de sensores e equipamentos finalmente puderam ser analisados, passando a gerar valor, até o ponto atual, em que o recente desenvolvimento das novas tecnologias de redes móveis permite que



sensores cada vez menores possam gerar uma quantidade gigantesca de dados com a interação entre os próprios equipamentos ou seu ambiente, pelo conceito de *internet das coisas* (do inglês *internet of things* – IoT). Segundo Cesar Taurion (2013, p. 29),

A internet das coisas vai criar uma rede de centenas de bilhões de objetos identificáveis e que poderão interoperar uns com os outros e com os *data centers* e suas nuvens computacionais. A internet das coisas vai aglutinar o mundo digital com o mundo físico, permitindo que os objetos façam parte dos sistemas de informação. Com a internet das coisas podemos adicionar inteligência à estrutura física que molda nossa sociedade.

O crescimento da capacidade de processamento, segundo a lei de Moore, deve dobrar a cada 18 meses. Inicialmente esse crescimento ocorria devido à miniaturização dos processadores. Atualmente a paralelização do processamento permitiu que esse crescimento se mantivesse. Espera-se que no futuro o desenvolvimento de novas tecnologias, como a computação quântica, possa aumentar a capacidade de processamento. Além disso, as novas tecnologias de armazenamento, como os discos de estado sólido (SSDs), têm permitido seu barateamento e aumento de capacidade.

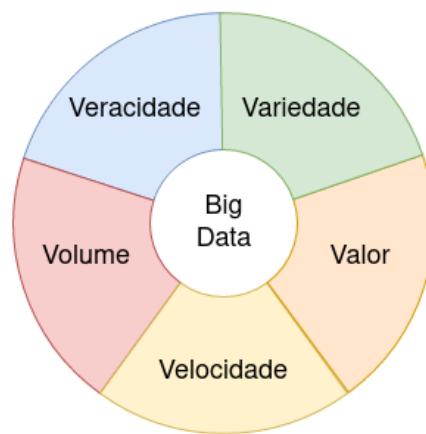
Podemos dizer que o aumento da geração de dados não deve ter seu ritmo reduzido tão cedo. Portanto, cada vez mais ferramentas poderosas são necessárias para analisar o imenso volume de dados gerados todos os dias no mundo.

TEMA 2 – OS Vs EM BIG DATA

Como acabamos de ver, o volume dos dados é um dos atributos mais relacionados ao conceito de Big Data, mas não é o único a defini-lo. Devemos considerar pelo menos outros dois aspectos: a velocidade em que os dados são processados e analisados, assim como a variedade dos dados, que podem ser obtidos de diversas fontes e se estruturar de diferentes formas.



Figura 1 – Os Vs em Big Data



2.1 Volume

O crescente volume de dados gerados a todo momento indica com relativa clareza que este é justamente o atributo mais significativo quando falamos em Big Data. No entanto, o que não fica muito claro é o ponto a partir do qual determinado conjunto de dados tem volume suficiente para ser considerado Big Data. Esse questionamento parte de uma premissa equivocada, pois o conceito de Big Data não pode ser definido única e exclusivamente pelo volume de dados processados. Um laboratório, por exemplo, pode necessitar de soluções de Big Data para visualizar imagens com 40 gigabytes, enquanto um observatório de astronomia pode necessitar de soluções de Big Data para analisar imagens e dados de sensores com terabytes de volume.

Portanto, o que define se um volume de dados necessita de soluções de Big Data não é seu tamanho, mas sua relação com a escalabilidade, eficiência, custo e complexidade. É o ponto em que a aplicação de soluções de Big Data supera os limites alcançados pelas tecnologias tradicionais que não foram projetadas para suportar esse volume.

2.2 Variedade

Um conjunto de dados de Big Data pode vir de diversas fontes e em diversos formatos. No entanto, as tecnologias tradicionais utilizam majoritariamente bancos de dados relacionais, ou seja, bancos que, embora



muito eficientes, são projetados para armazenar dados previamente estruturados, respeitando a propriedade Acid para que a integridade dos dados seja garantida da seguinte forma:

- **Atomicidade**: garante que transações não tenham atualizações parciais, ou seja, devem se comportar de forma indivisível e ser feitas por inteiro, ou então não são feitas;
- **Consistência**: garante que transações não afetem a consistência do banco. Dessa forma, as transações são completadas apenas se não ferirem nenhuma regra de integridade do banco, levando o banco de dados de um estado consistente a outro estado também consistente;
- **Isolamento**: garante que transações concorrentes não interfiram nos eventos umas das outras. As transações devem ter o mesmo resultado, como se fossem executadas uma após a outra;
- **Durabilidade**: garante que todos os efeitos de uma transação completada com sucesso persistam mesmo na ocorrência de falhas externas.

No entanto, estima-se que a maioria dos dados existentes seja de dados não estruturados ou semiestruturados (Marquesone, 2016). Os dados semiestruturados compreendem estruturas previamente definidas, mas que não exigem o mesmo rigor que os bancos de dados relacionais – é o caso de arquivos nos formatos JavaScript Object Notation (JSON) ou eXtensible Markup Language (XML). Já os dados não estruturados são todos aqueles excessivamente complexos para serem armazenados apenas com ferramentas tradicionais de armazenamento e gerenciamento de dados. É o caso de mídias como vídeos, imagens, áudios e até mesmo alguns formatos de texto.

Os dados não estruturados ou semiestruturados exigem que a tecnologia adotada forneça uma estrutura flexível o bastante para que dados tão diversos possam ser analisados. Além disso, a estrutura deve permitir a utilização de ambientes distribuídos, incluindo a variedade de soluções e tecnologias necessárias para atender a demanda específica que a solução requer.

Há uma variedade muito grande de dados em variados formatos, sendo utilizados por uma variedade também muito grande de soluções com necessidades muito específicas. A solução de Big Data deve ser capaz de integrar e interagir com toda essa diversidade de dados.



2.3 Velocidade

Devido à popularização da internet, das mídias e redes sociais, dos dispositivos móveis e da internet das coisas, dados são gerados de forma cada vez mais rápida e por cada vez mais agentes. Por exemplo, um único carro moderno pode ter cerca de 100 sensores, que geram dados a cada instante. Por isso a velocidade em que esses dados são coletados, analisados e utilizados se torna cada vez mais importante.

Dados perdem valor com o tempo; por exemplo, sites de comércio eletrônico que atualizam seus preços de acordo com a demanda de suprimentos podem maximizar as vendas. Outro exemplo são os serviços de tráfego, que podem oferecer melhores rotas ou sistemas críticos que dependem das informações geradas em tempo real. Portanto, para muitos serviços, de nada adianta ter a capacidade de processar um volume imenso de dados, de diferentes lugares e formas, se a solução não for capaz de responder no tempo necessário para que a informação seja útil e não perca seu valor.

2.4 Valor

Ao definir o conceito de Big Data, alguns autores fazem referência a apenas três fatores (Vs): **volume, variedade e velocidade**. Reunidos, definem Big Data como a coleta e análise de um volume imenso de dados que podem vir de diversas fontes e ter uma grande variedade de formatos, numa velocidade altíssima. No entanto, podemos avaliar também quão valioso e significativo um dado pode ser para uma solução. Dessa forma, temos como saber quais dados podem gerar maior valor para a solução e, por isso, devem ser priorizados. Ao priorizar ou escolher os dados corretos, é possível otimizar a solução para que o valor gerado seja o mais adequado.

2.5 Veracidade

Outro fator de grande importância é saber quão confiável é o conjunto de dados que estamos utilizando. Isso impacta diretamente na confiabilidade da informação extraída. Estima-se que dados de baixa qualidade custem à economia trilhões de dólares anualmente. Dados com uma precisão inadequada



ou falsos podem levar a informações incorretas e até inviabilizar soluções. Portanto, se a veracidade dos dados coletados não for avaliada e garantida, corremos o risco de afetar o valor e a validade das informações que geramos.

2.6 Temos uma definição para o conceito de Big Data?

Dado o que consideramos, a definição de Big Data pode variar bastante. Alguns autores podem resumir a definição nos três principais fatores (volume, variedade e velocidade), enquanto outros chegam a utilizar até dez fatores (ou mais). No entanto, não divergem muito dos 5 Vs que vimos até aqui. Muitas vezes acontece de alguns autores aglutinarem algumas ideias em um dos fatores ou as subdividem em novos fatores que podem ser mais adequados à solução específica que se está desenvolvendo.

Portanto, para nossos propósitos, vamos definir Big Data como o conjunto de práticas e técnicas que envolvem a coleta e análise de um grande volume de dados confiáveis e variados (tanto no formato quanto na origem), com a velocidade necessária para gerar o valor adequado à solução. O que realmente importa é o valor que podemos gerar quando nos livramos das limitações relativas ao volume, variedade, velocidade e veracidade dos dados processados.

TEMA 3 – OBTEÇÃO E ARMAZENAMENTO DE DADOS

No que diz respeito ao Big Data, tudo se inicia com a obtenção dos dados que serão processados. Como vimos, os dados podem vir de diferentes origens e ter diferentes formatos. Pode ser que os dados necessários ainda não existam e precisem ser gerados, que sejam internos à própria aplicação ou que devam ser buscados de fontes externas. Todas essas questões fazem parte da fase de obtenção de dados e, para isso, estratégias de como os dados serão coletados e armazenados devem ser definidas.

3.1 Obtenção de dados

A obtenção de dados pode ser compreendida com diferentes estratégias de captura e utilização de dados no projeto.



- **Dados internos:** são os dados que o proprietário do projeto (empresa) já tem e cujo controle já detém. Tais dados podem vir de sistemas de gerenciamento da própria empresa, como: sistemas de gerenciamento de projetos; automação de marketing; sistemas *customer relationship management* (CRM); sistemas *enterprise resource planning* (ERP); sistemas de gerenciamento de conteúdo; dados do departamento de recursos humanos; sistema do gerenciamento de talentos; procurações; dados da intranet e do portal da empresa; arquivos pertencentes à empresa, como documentos escaneados, formulários de seguros, correspondências, notas fiscais, entre outros; documentos gerados por colaboradores, como planilhas em XML, relatórios em PDF, dados em CSV e JSON, e-mails, documentos de texto em diversos formatos, apresentações e páginas web; e registros de log de eventos, de dados de servidores, logs de aplicações ou de auditoria, localização móvel, logs sobre o uso de aplicativos móveis e logs da web;
- **Dataficação:** é a transformação de ações sociais em dados quantificados de forma a permitir o monitoramento em tempo real e análises preditivas;
- **Dados de sensores:** são dados inseridos no contexto da internet das coisas, em que os objetos se comunicam com outros objetos e pessoas. Para esse tipo de solução, é necessário prover um meio de transmitir dados entre os sensores e um servidor capaz de armazenar os dados das interações. A obtenção de dados de sensores ocorre em tempo real, por isso os maiores desafios estão no volume e na velocidade com que os dados são gerados. Exemplos de dados de sensores são aqueles coletados de medidores inteligentes, sensores de carros, câmeras de vigilância, sensores do escritório, maquinários, aparelhos de ar-condicionado, caminhões e cargas;
- **Dados de fontes externas:** são dados obtidos de domínio público, como dados governamentais, dados econômicos, censo, finanças públicas, legislações, entre outros. Podemos considerar também qualquer tipo de dados obtidos por sites de terceiros, como mídias e textos de sites da web, além dos dados obtidos de mídias sociais. São basicamente todos os dados possíveis de obter por requisições na web ou uma *application programming interface* (API) dedicada. Muitos desses dados são



disponibilizados por APIs acessadas via *representational state transfer* (Rest), obtendo-se os dados requisitados em formato JSON.

3.2 Armazenamento

Como vimos, os bancos de dados relacionais foram por muito tempo o padrão mundial de armazenamento. Nesse modelo os dados são armazenados de forma previamente definida em estruturas de tabelas que podem ser relacionadas com outras tabelas da mesma base de dados. Uma das características mais importante desse tipo de armazenamento é o suporte a transações Acid.

Outra característica importante é o uso de *structured query language* (SQL) para operações de criação e manipulação de dados, o qual permitiu que os dados armazenados tivessem sua integridade garantida, e também permitiu gerar consultas mais complexas. No entanto, o crescimento constante na geração de dados mostrou os limites dos bancos de dados relacionais como única solução de armazenamento, principalmente no que se refere à **escalabilidade, disponibilidade e flexibilidade**.

3.2.1 Escalabilidade

Uma solução é considerada escalável quando mais carga é adicionada e mesmo assim o desempenho se mantém adequado. Com determinado volume de dados, os bancos de dados tradicionais conseguem manter esse desempenho apenas ao adicionar mais recursos computacionais à infraestrutura, o que é conhecido como *escalabilidade vertical*. No entanto, o volume de dados necessários aumentou tanto que esse tipo de solução não é mais viável em todos os casos, uma vez que os custos de tais recursos podem ser muito elevados.

3.2.2 Disponibilidade

Para que um serviço seja considerado de alta disponibilidade, o tempo em que ele se mantém operando deve ser priorizado em comparação às demais propriedades Acid. Portanto, deve-se garantir que o serviço se mantenha operando mesmo em casos de falha na infraestrutura.



3.2.3 Flexibilidade

Um serviço flexível é aquele capaz de comportar uma grande diversidade de dados. O grande problema desse requisito é que muitas vezes é inviável modelar um conjunto de dados de forma antecipada e que conte coleste características não estruturadas.

Concluímos que os bancos de dados tradicionais já não são a solução mais adequada para suprir os requisitos exigidos em soluções de Big Data. Dessa forma, soluções alternativas foram criadas para atender a esse tipo de demanda.

3.3 NoSQL

A noção de NoSQL incorpora uma ampla variedade de tecnologias de bancos de dados desenvolvidos como resposta à demanda de aplicações modernas. Quando comparadas com bancos de dados relacionais, bancos NoSQL são mais escaláveis, têm melhor desempenho, e seu modelo de dados resolve questões que os bancos de dados relacionais não foram projetados para resolver, como grandes volumes de dados de estruturados, semiestruturados e não estruturados que se modificam rapidamente, arquiteturas distribuídas geograficamente, entre outras. Os modelos de bancos de dados NoSQL podem ser classificados de acordo com a estrutura em que os dados são armazenados. Existem vários modelos, e os quatro principais são: **orientado a chave-valor**, **orientado a documentos**, **orientado a colunas** e **orientado a grafos**.

3.3.1 Bancos de dados orientados a chave-valor

Os bancos de dados orientados a chave-valor são os modelos mais simples de NoSQL. Cada item é armazenado como um atributo-chave normalmente composto de um campo tipo *string* associado a um valor que pode conter diferentes tipos de dados. Esse modelo não exige um esquema predefinido, como acontece nos bancos de dados relacionais. Esse tipo de banco de dados pode ser utilizado tanto para armazenar os dados quanto para mantê-los em *cache* para agilizar o acesso. Portanto, é um tipo de banco muito importante para aplicações que realizam muitos acessos aos dados. Apesar das



vantagens dos bancos de dados chave-valor, ele tem limitações. A única forma de realizar consultas é por meio da chave, uma vez que não é possível indexar utilizando o campo valor.

3.3.2 Bancos de dados orientados a documentos

Os bancos de dados orientados a documentos são uma extensão dos bancos de chave-valor, uma vez que também associam uma chave a um valor. Mas nesse caso o valor é necessariamente uma estrutura de dados chamada *documento*. A noção de documento é o conceito central desse tipo de banco de dados, e consiste em estruturas de um padrão definido, tal como XML, YAML, JSON, ou até formatos binários.

O documento pode se comportar de forma muito semelhante ao conceito de *objeto* em programação. Além disso, permite-se um conjunto de operações muito semelhantes ao padrão Crud: *creation* (inserção), *retrieval* (busca, leitura ou requisição), *update* (edição ou atualização), e *deletion* (remoção, deleção). Isso permite criar consultas e filtros sobre os valores armazenados, e não somente pelo campo-chave.

Outra característica desse banco é a alta disponibilidade, uma vez que permite trabalhar com a replicação de dados em *cluster*, garantindo que o dado ficará disponível mesmo em caso de falha no servidor.

3.3.3 Bancos de dados orientados a colunas

Os bancos de dados orientados a colunas são otimizados para buscas em grandes bancos de dados. Podem ser interpretados como bancos chave-valor bidimensionais; neles, o que seria uma tabela num banco de dados relacional seria um item identificado por uma chave associada a um valor que pode conter vários conjuntos de chave-valor.

Tais conjuntos são o equivalente ao campo de uma coluna de determinado item. Isso permite flexibilidade, tal que cada registro pode ter um número diferente de colunas. Os bancos orientados a colunas também podem ter o conceito de famílias de colunas. Cada família tem múltiplas colunas que são utilizadas em conjunto, de maneira semelhante às tabelas dos bancos de



dados relacionais. Dentro de uma família de colunas, os dados são armazenados linha por linha, de forma que as colunas de uma linha sejam armazenadas juntas.

Esse banco de dados é altamente adequado a soluções que necessitam trabalhar com volumes imensos de dados, alto desempenho, alta disponibilidade no acesso, armazenamento de dados e flexibilidade na inclusão de campos. Além disso, sua solução tolera eventuais inconsistências.

3.3.4 Bancos de dados orientados a grafos

Os bancos de dados orientados a grafos são muito úteis quando as relações entre os dados são mais importantes que os dados em si. Esse tipo de banco é utilizado para armazenar a informação sobre as redes de dados. Em vez de os dados serem formatados em linhas e colunas, são estruturados em vértices, arestas, propriedades para armazenar os dados coletados e os relacionamentos entre esses dados.

Os vértices representam entidades ou instâncias. Equivalem a um registro, uma linha dos bancos de dados relacionais, ou um documento num banco de dados orientado a documentos. As arestas (ou relações) são as linhas que conectam os vértices, representando a relação entre os dois vértices conectados. As arestas podem ser direcionais ou não direcionais, e propriedades são informações relacionadas com os vértices.

As soluções NoSQL não foram desenvolvidas para substituir os bancos de dados relacionais, mas para complementá-los. A tendência é adotar soluções híbridas, em que cada banco é utilizado onde possa ser mais adequado.

3.4 Governança de dados

Para qualquer empresa que adote estratégias de Big Data em suas soluções, é muito importante gerenciar dados de forma que seu uso seja o mais eficiente e confiável possível. A governança de dados inclui as pessoas, os processos e as tecnologias necessárias para proteger os ativos de dados da companhia, de forma a garantir que os dados da empresa sejam comprehensíveis, corretos, completos, confiáveis, seguros e detectáveis. De acordo com Marquesone (2016), os principais tópicos na governança de dados são:



- **Arquitetura dos dados:** é o que define o modelo para gerenciar ativos de dados, alinhando-se à estratégia da empresa para estabelecer requisitos e projetos de dados estratégicos que atendam a esses requisitos. Todas as políticas que padronizam os elementos de conjuntos de dados, os protocolos e boas práticas são criadas para garantir a adoção dos padrões definidos;
- **Auditoria:** comprehende que os dados devem permitir o próprio rastreio, e deve ser possível conhecer quando os dados foram criados, como estão sendo utilizados e quais os seus impactos;
- **Metadados:** são dados a respeito de outros dados. No contexto da governança de dados, é o que permite a visualização holística e açãoável de uma cadeia de suprimentos de informação. É o que permite gerenciar as alterações nos dados, a auditoria e rastreabilidade do fluxo de dados, e também a melhora na acessibilidade dos dados por buscas e mapas visuais;
- **Gerenciamento de dados-mestre (*master data management – MDM*):** dados-mestre são aqueles essenciais para o negócio de uma empresa. Podemos compreendê-los como o estabelecimento e gerenciamento de dados no nível organizacional, que fornecem dados-mestre precisos, consistentes e completos por toda a empresa e para parceiros de negócio;
- **Modelagem dos dados:** como vimos, é importante que toda a variedade de dados disponíveis seja modelada, de forma a permitir a utilização de padrões de dados, evitar redundâncias, definir como os dados serão utilizados, e encontrar as melhores formas de construir uma arquitetura de dados mais ágil e governável;
- **Qualidade dos dados:** comprehende os processos incorporados com o objetivo de aperfeiçoar a qualidade dos dados de forma que contenham menos erros, estejam mais completos e possam otimizar a utilidade dos dados. Inclui a criação de *profiles* de dados, estratégias de limpeza, filtragem e agrupamento de dados;
- **Segurança:** relaciona-se à gestão de risco relacionado à coleta, armazenamento, processamento e análise dos dados. Isso implica que todos os dados importantes e sensíveis devem ser utilizados de maneira correta e segura, de forma a prevenir o mau uso em todos os níveis. Pode-



se utilizar estratégias de criptografia, definição e proteção a dados sensíveis, políticas de proteção de integridade, disponibilidade, confiabilidade e autenticidade dos dados.

A governança de dados tem se tornado ainda mais importante com a adoção cada vez maior de soluções de Big Data, uma vez que a veracidade e o valor dos dados são diretamente influenciados por seus processos, permitindo a criação de modelos de negócios mais inovadores, confiáveis e eficientes.

TEMA 4 – PROCESSAMENTO DE DADOS

Tão logo os dados são capturados e armazenados, inicia-se a fase de processamento dos dados. Para isso, devemos avaliar algumas questões relacionadas ao processamento, como alocação de recursos, escalabilidade, disponibilidade, desempenho e o tipo de processamento.

4.1 Escalabilidade

Uma das questões mais importantes quando tratamos de um volume de dados que pode crescer imensamente é a escalabilidade. Um sistema escalável é aquele em que o desempenho não se deteriora com o aumento significativo de dados sendo processados. A capacidade de processamento da plataforma deve escalar proporcionalmente à demanda. Para isso, é necessário monitorar a execução de forma a impedir o esgotamento de recursos.

Outro aspecto importante é que não se deve sacrificar a disponibilidade da plataforma. Mesmo que ocorram falhas, o serviço deve manter-se ativo. No que se refere à escalabilidade, existem duas estratégias possíveis. Podemos adotar um modelo de **escalabilidade vertical** ou de **escalabilidade horizontal**.

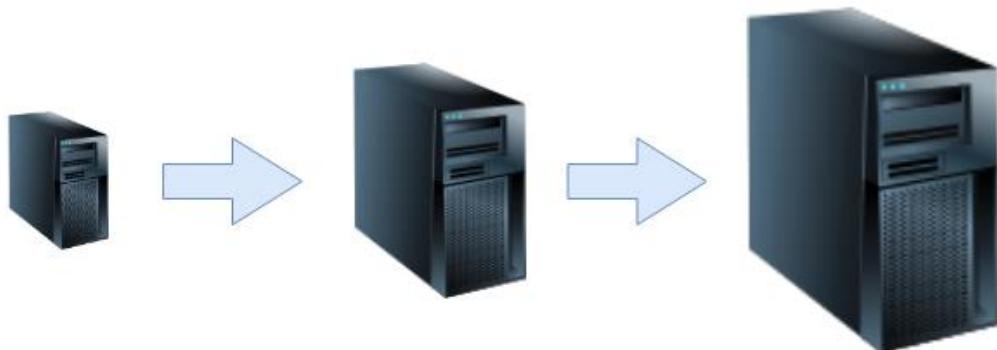
4.1.1 Escalabilidade vertical

A escalabilidade vertical se refere à adição de capacidade de processamento de um único recurso com a atualização da infraestrutura. Aumenta-se a capacidade de processamento da plataforma pela atualização da infraestrutura.



Esse tipo de estratégia costuma comprometer a disponibilidade do serviço, a não ser que haja redundância na infraestrutura. Sua vantagem é não exigir modificações nos algoritmos, mas em geral é uma solução que não atende à demanda quando aplicada ao contexto em que o volume de dados cresce rapidamente, como é o caso de soluções de Big Data.

Figura 2 – Escalabilidade vertical



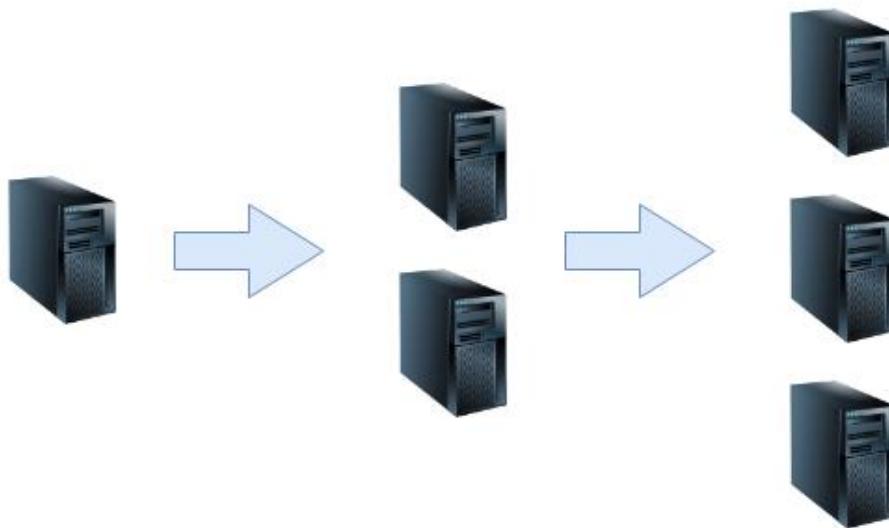
4.1.2 Escalabilidade horizontal

Uma estratégia muito mais adequada é a escalabilidade horizontal. Com ela o processamento é distribuído num conjunto de tarefas menores para serem processadas num *cluster* de recursos, de forma que o aumento da quantidade de recursos computacionais seja capaz de suprir o aumento na demanda de processamento. Além disso, os recursos do *cluster* se comportam de forma independente e redundante, colaborando com a disponibilidade da solução.

Dessa forma, o custo de melhorias na infraestrutura é reduzido, pois não é necessário interromper sua operação. A redistribuição de carga entre os recursos de um *cluster* é uma forma simples de regular a capacidade de processamento de acordo com a demanda; essa característica confere uma imensa capacidade de escalabilidade. Sua desvantagem é que a tecnologia tradicional não foi projetada para esse tipo de estratégia. Portanto, geralmente requer reimplementação de software para utilizar tecnologias de sistemas distribuídos.



Figura 3 – Escalabilidade horizontal



4.2 Processamento de dados com Hadoop

Originalmente, o Hadoop foi projetado para funcionar como um motor de busca de código aberto. O framework Hadoop se baseia em duas tecnologias que visam o suporte ao armazenamento e processamento distribuído de grandes volumes de dados:

- O sistema de arquivos distribuído Hadoop Distributed File System (HDFS);
- O modelo de programação distribuída MapReduce.

As principais características do Hadoop que envolvem o processamento de grandes volumes de dados são:

- **Baixo custo:** por ser projetado para utilizar em servidores tradicionais, não exige a implantação de hardware específico;
- **Escalabilidade:** devido à adoção de tecnologias distribuídas, sua capacidade de processamento escala linearmente. Isso significa que o aumento de recursos de computação se reflete diretamente na capacidade de processamento. E não é necessário alterar a base de código cada vez que a infraestrutura é atualizada;



- **Tolerância a falhas:** seu modelo de escalabilidade horizontal garante que, mesmo que algum dos recursos apresente falhas, os recursos restantes supram a demanda;
- **Balanceamento de carga:** a tecnologia de processamento distribuído evita gargalos que podem limitar o processamento de recursos. Dessa forma, todos os recursos operam de forma otimizada;
- **Comunicação entre máquinas e sua alocação:** ocorre de forma transparente para o usuário.

Todas essas características são implementadas pelo Hadoop e permitem que o desenvolvedor concentre seus esforços na lógica do problema. Dessa forma, a análise de um grande conjunto de dados – anteriormente ignorados devido a custos inviabilizantes – foi permitida com o surgimento das tecnologias distribuídas HDFS e Hadoop MapReduce.

O sistema de arquivos distribuídos HDFS foi criado para gerenciar o armazenamento das máquinas do *cluster*. Ele tem escalabilidade para armazenar grandes volumes de dados de forma tolerante a falhas, com recuperação automática. A disponibilidade é garantida pela replicação de dados, e o sistema se encarrega de quebrar o arquivo em blocos menores, replicando-os algumas vezes em diferentes servidores.

O Hadoop MapReduce foi projetado para gerenciar o processamento de dados distribuído com a divisão de uma aplicação em tarefas independentes executadas em paralelo nos servidores do *cluster*. O processamento é dividido nas seguintes etapas:

- **Map:** recebe uma entrada de dados e retorna um conjunto de pares no formato de pares chave-valor. As operações dessa etapa são definidas pelo desenvolvedor;
- **Shuffle:** os dados retornados pelo Map são organizados de forma a aglutinar todos os valores associados a uma única chave. Para cada chave teremos um par que a associa com uma lista contendo todos os valores relacionados a essa chave como valor. Essa etapa é feita automaticamente;
- **Reduce:** os dados organizados são recebidos, e operações definidas pelo desenvolvedor são realizadas, gerando o resultado da aplicação.



4.3 Processamento em tempo real

Apesar de todas as vantagens do Hadoop, ele não é adequado a todas as soluções de Big Data, uma vez que foi desenvolvido para processamento em lote. Isso significa que primeiramente são formados grupos de dados coletados num período de tempo, para só então os dados serem processados. Desde que os dados tenham sido gerados até seus resultados serem processados e, então, respondidos, temos uma quantidade de tempo significativa. Além disso, para muitos casos, o processamento dos dados deve se dar de forma contínua. No entanto, no modelo em lote, o processamento se encerra tão logo os resultados são retornados.

Diferentemente, muitas aplicações precisam que os dados sejam processados à medida que chegam à aplicação – ou seja, em tempo real –, e cada item de dados que chega à aplicação é processado imediatamente. Para isso, o processamento em tempo real tem alguns requisitos importantes:

- **Baixa Latência:** o tempo de processamento de um item de dado deve ser no máximo igual ao tempo em que novos dados chegam ao fluxo;
- **Consistência:** a solução deve ser capaz de operar com imperfeições e manipular inconsistências;
- **Alta disponibilidade:** etapas de coleta, transmissão e processamento de dados podem causar grandes impactos se ficarem indisponíveis, resultando na perda de dados significativos para a aplicação.

O processamento em tempo real é importante para soluções web com o rastreamento de usuários e análises de preferências, detecção de fraudes, redes sociais, com a identificação de tendências, além da internet das coisas, com milhares de objetos e sensores que geram dados o tempo todo.

Se soluções de processamento em lote, como o Hadoop, tiverem dificuldades em atender às demandas de velocidade necessárias para o processamento em tempo real, precisamos de uma solução que faça o processamento de fluxos de dados.



4.4 Processamento de dados com Spark

Spark é uma tecnologia que tem se destacado no processamento em tempo real, devido ao seu desempenho. Trata-se de um framework que estende o modelo de programação MapReduce, otimizando o desempenho em programação distribuída. O Hadoop compreende tanto um componente de armazenamento – o HDFS – quanto um componente de processamento – o Hadoop MapReduce. No entanto, o Spark concentra seus esforços no processamento de dados, podendo muitas vezes ser utilizado com o Hadoop, uma vez que seu processamento costuma ter desempenho muito superior ao Hadoop MapReduce.

Os principais componentes do Spark são:

- **Spark Core:** disponibiliza as funções básicas para o processamento, como Map, Reduce, Filter, Collect, entre outras;
- **GraphX:** realiza o processamento sobre grafos;
- **SparkSQL:** para a utilização de SQL em consultas e processamento sobre dados;
- **Mlib:** disponibiliza a biblioteca de aprendizado de máquina.

O Spark não conta com um sistema próprio de gerenciamento de arquivos, portanto, precisa ser integrado a um, como o HDFS do Hadoop, como foi sugerido. Mas também é possível utilizá-lo com uma base de dados em *cloud computing*. A arquitetura Spark é definida no Spark Core e é composta principalmente de três componentes principais:

- **Driver Program:** aplicação principal que gerencia a criação e executa o processamento definido pelo programador;
- **Cluster Manager:** administra o *cluster* de máquinas quando a execução for distribuída;
- **Workers:** executam as tarefas enviadas pelo Driver Program.

Os conceitos mais importantes utilizados na programação e no desenvolvimento de soluções com Spark incluem:



- **Resilient Distributed Dataset (RDD)**: funciona como uma abstração do conjunto de objetos distribuídos pelo *cluster*. É o objeto principal do modelo de programação no Spark;
- **Operações**: são as transformações e ações realizadas num RDD;
- **Spark Context**: objeto que representa a conexão da aplicação com o *cluster*. Pode ser utilizado para criar RDDs, acumuladores e variáveis no *cluster*.

TEMA 5 – ANÁLISE E VISUALIZAÇÃO

Apenas recentemente a capacidade de armazenamento e processamento se tornaram suficientes para permitir que dados antes ignorados fossem analisados. Entretanto, além dos componentes tecnológicos, o analista de dados deve ser capaz de identificar quais dados se deve utilizar, como integrá-los, quais perguntas serão úteis para a tomada de decisão, e qual a melhor maneira de apresentar os resultados obtidos da análise.

5.1 Análise de dados

A extração de informações úteis pela análise de dados não é uma tarefa simples. Em muitos casos, os dados podem ter informações incompletas, inconsistências, caracteres indesejados, estarem corrompidos, duplicados, em formato inadequado, e outros tipos de problema. Segundo Marquesone (2016), cerca de 80% do tempo da análise de dados é utilizado apenas para limpar e preparar os dados.

Durante a análise, podemos constatar a importância da qualidade dos dados utilizados pois, sem um processo de inspeção, muitos dados incorretos podem ser descartados. No entanto, mesmo que a qualidade seja garantida, a busca por padrões nos dados ainda é um grande desafio. É muito fácil analisá-los de forma errada, ao não se identificar corretamente relações de correlação e causalidade, e propagar erros que invalidem toda a análise. Por fim, é necessário validar todos os resultados gerados pela análise dos dados antes de serem utilizados.



5.2 O processo de análise de dados

A análise de dados inclui como atividades a identificação de padrões nos dados, sua modelagem e classificação, detecção de grupos, entre muitas outras. Para isso, utilizamos técnicas matemáticas, estatísticas e de aprendizado de máquina. O aprendizado de máquina pode ser muito útil na automatização da construção de modelos analíticos. Pode-se extrair informações úteis e padrões ocultos em conjuntos massivos de dados.

Os processos de análise de dados podem ser definidos de diversas formas. Uma delas seria por um padrão aberto conhecido pela sigla CRISP-DM (*cross-industry standard process for data mining*), que define as seguintes fases e tarefas:

- **Entendimento de negócio:** determinar os objetivos de negócio, seu contexto e critérios de sucesso; avaliar recursos disponíveis, riscos e contingências, definir terminologias e calcular custos e benefícios; determinar os objetivos da mineração de dados e seus critérios de sucesso; e produzir um plano de projeto. Nessa fase são definidas as perguntas, os objetivos e os planos;
- **Compreensão dos dados:** fazer a coleta inicial e descrever os dados; fazer análise exploratória; e verificar a qualidade dos dados. O objetivo é entender a estrutura, atributos e contexto em que os dados estão inseridos;
- **Preparação dos dados:** descrever o conjunto, selecionar, filtrar e limpar os dados, e minimizar a geração de resultados incorretos; construir dados (atributos derivados, registros gerados); integrá-los (mesclagem ou redução); formatá-los e estruturá-los;
- **Modelagem dos dados:** selecionar técnicas de modelagem; projetar testes; definir e construir o modelo de dados, seus parâmetros e sua descrição; e validar o modelo e definir os parâmetros a revisar. Para construir o modelo, utilizamos tarefas de algoritmos de extração de padrões que podem ser agrupadas como atividades descritivas ou preditivas;



- **Avaliação do modelo:** avaliar os resultados do modelo; revisar processos; e determinar os passos seguintes. Avalia-se a precisão dos resultados gerados com os modelos de dados;
- **Utilização do modelo:** planejar a entrega; planejar o monitoramento e a manutenção; produzir relatório final; e documentar e revisar o projeto. Os modelos aprovados são então utilizados e monitorados.

5.3 Visualização de dados

Obtidos os resultados pela análise dos dados, ou até antes disso, é interessante poder comunicar as informações obtidas. Ao mesmo tempo, é importante entender quais informações são mais relevantes e qual a melhor forma de apresentá-las com clareza. Uma vez que nós, como humanos, somos dotados de grande percepção visual, a representação dos dados de forma gráfica é muito eficiente para expressar as informações que obtivemos dos dados. Assim, a visualização de dados é definida como **a comunicação da informação utilizando representações gráficas**.

5.3.1 Visualização exploratória

A análise de dados requer que eles sejam avaliados de forma detalhada. Para melhorar a compreensão deles nessa etapa, é possível usar a visualização exploratória, que auxilia na identificação de estruturas das variáveis, das tendências e de relações, permitindo até mesmo a detecção de anomalias nos dados.

Existem muitas formas de representar dados graficamente. Cada uma delas é capaz de exibir certo nível de detalhamento ou destacar características específicas. Inclusive é muito comum que o analista de dados use diferentes tipos de gráfico para estruturar os dados conforme necessário, para melhorar sua compreensão.

5.3.2 Visualização explanatória

Quando o analista já tiver resultados concretos, ele está pronto para comunicar suas percepções. Essa etapa é definida como *visualização explanatória*. Durante ela, o interesse do analista é destacar os detalhes



importantes, de forma a comunicar os resultados obtidos de um grande volume de dados em informações mais concisas e de fácil compreensão no formato de uma interface visual. Em muitos casos, essa informação permite revelar tendências e desvios que podem servir de apoio a tomadas de decisão.

Uma interface visual permite que o leitor veja características específicas e detalhadas dos dados. Para isso, existem muitos atributos que devem ser analisados durante a criação dos gráficos. Cada tipo de gráfico pode ser mais adequado para comunicar certa informação a respeito de seus dados. Por exemplo, gráficos de colunas, barras, áreas circulares, linhas e de dispersão são mais adequados para comparar valores, enquanto gráficos de dispersão, histogramas e gráficos de área são mais adequados para destacar a distribuição de um conjunto de dados. Para cada necessidade, existe algum tipo de gráfico apropriado.

5.4 A visualização de dados

A literatura mostra que existem sete etapas para a visualização de dados, incluindo algumas que fazem parte das etapas de coleta, armazenamento, processamento e análise de dados.

1. **Aquisição:** etapa em que ocorre a coleta de dados;
2. **Estruturação:** define-se a estrutura em que os dados são padronizados;
3. **Filtragem:** dados incorretos, incompletos ou desinteressantes para a análise são removidos;
4. **Mineração:** parte da etapa de análise de dados que extrai informações dos dados;
5. **Representação:** etapa da análise exploratória que gera um modelo visual básico de dados;
6. **Refinamento:** técnicas gráficas para tornar a visualização mais eficiente;
7. **Interação:** inclui funcionalidades que oferecem melhor experiência para o leitor.

Existem ferramentas que auxiliam a visualização de dados a ponto de automatizar muitas etapas. Como vimos, ela pode ser importante durante a análise dos dados, pois contribui para resultados com maior precisão, ou ainda atende soluções que necessitam de visualização em tempo real.



FINALIZANDO

Neste capítulo, vimos uma introdução aos principais fundamentos que compõem o conceito de Big Data. Iniciamos com uma breve contextualização histórica da evolução da geração de dados e dos avanços tecnológicos que aumentaram o volume de dados a uma escala imensa, permitindo que dados anteriormente ignorados ou descartados passassem a ser analisados de forma cada vez mais detalhada.

Vimos que uma solução Big Data pode ser definida pela coleta, processamento, análise e visualização de um volume muito grande de dados confiáveis (quanto à veracidade) e variados (tanto no formato quanto na origem), com a velocidade necessária para gerar o valor adequado à solução. Vimos um pouco sobre os processos de coleta, armazenamento, processamento, análise e visualização de dados, passando por noções básicas de bancos de dados não relacionais (NoSQL), Hadoop, Spark, entre outros.



REFERÊNCIAS

TAURION, C. **Big Data**. Rio de Janeiro: Brasport, 2013.

MARQUESONE, R. **Big Data**: técnicas e tecnologias para extração de valor dos dados. São Paulo: Casa do Código, 2016.



BIG DATA

AULA 3

Prof. Luis Henrique Alves Lourenço



CONVERSA INICIAL

Nesta aula, aprofundaremos a discussão sobre os componentes Hadoop responsáveis pela integração da ferramenta de Big Data com Bancos de Dados Relacionais, Bancos de Dados Não Relacionais, também conhecidos como *Bancos NoSQL*. Além disso, vamos conhecer as ferramentas capazes de realizar consultas aos dados independentemente de onde eles se situam, inclusive misturando os de armazenamento de forma transparente ao usuário.

TEMA 1 – HIVE

A primeira característica que podemos destacar a respeito do Hive é que ele se aproveita de uma sintaxe semelhante ao SQL. O Hive foi projetado para ser uma aplicação de *Data Warehouse open source* que opera sobre componentes do Hadoop, como HDFS e MapReduce. *Data Warehouse* é uma categoria de aplicações responsáveis por armazenar dados de diversos sistemas em um repositório único. Os dados de tais sistemas são transformados para serem formatados de acordo com um padrão específico. Dessa forma, o Hive é capaz de traduzir consultas em um dialeto de SQL, o Hive SQL, para tarefas do MapReduce ou Tez. Por sua semelhança com consultas SQL, o HiveQL é uma ferramenta muito poderosa e fácil de utilizar.

O Hive permite também a requisição de consultas de maneira interativa por meio de um prompt de comando ou um terminal, assim como qualquer banco de dados relacional. Uma vez que o Hive opera sobre uma estrutura de Big Data, ele é altamente escalável e adequado para trabalhar com grandes volumes de dados.

Por essas características, o Hive é um componente muito adequado para ferramentas OLAD (*OnLine Analytical Processing*), utilizadas para analisar dados multidimensionais interativamente por diversas perspectivas, o que permite consultas analíticas complexas e especializadas com rápido tempo de execução. Sem o Hive, isso é possível apenas por meio de implementações de funções MapReduce em Java, sendo muito mais complexo do que as consultas em HiveQL.

É válido destacar que os dados utilizados em consultas HiveQL não estão em um banco de dados relacionais com tabelas bem definidas. Portanto,



algumas das funcionalidades comuns de bancos de dados não existem da mesma maneira. Por exemplo, as consultas não realizam transações e, por isso, operações como *update*, *insert* e *delete* não funcionam da mesma forma. Portanto, podemos dizer que a grande desvantagem do uso de Hive é a sua limitação quanto às operações possíveis. Componentes como Pig e Spark oferecem uma quantidade maior de recursos e permitem realizar operações mais complexas.

Ainda assim, HiveQL implementa grande parte das operações SQL e algumas extensões. Uma das possibilidades permitidas pelo HiveQL é o armazenamento do resultado de uma consulta em uma estrutura chamada de *view*, que possibilita que consultas futuras acessem tais dados como se estivessem em tabelas de um banco. Além disso, o Hive permite as quatro operações básicas de transações em bancos de dados definidas pelas propriedades ACID: Atomicidade, Consistência, Isolamento e Durabilidade.

1.1 Arquitetura Hive

O Hive oferece a possibilidade de implementar extensões por meio de funções definidas pelo usuário, pelo servidor Thrift (para C++, Python, Ruby, e outras linguagens) e por drivers JDBC e ODBC. Além disso, é possível efetuar consultas por uma interface de linha de comando (CLI), ou pela *Hive Web Interface (HWI)*.

Todos os comandos e consultas são recebidos pelo *Driver*, que compila a entrada, otimiza a computação necessária e executa os passos, originalmente com tarefas MapReduce. O Hive não gera funções em Java para o MapReduce. Em vez disso, ele utiliza Mappers e Reducers genéricos, sequenciados por planos de trabalho escritos em XML. Dessa forma, o Hive se comunica com a aplicação mestre para iniciar a execução pelo MapReduce ou Tez.

Bancos de Dados Relacionais originalmente foram projetados baseados em uma técnica para o armazenamento de dados conhecida por *schema on write*. Essa estratégia define que as partes dos dados precisam se ajustar a um padrão ou um plano no momento da escrita. Com o tempo, uma característica dessa estratégia se tornou evidente: ela é muito restritiva. Perde-se muito tempo ajustando os dados à estrutura definida. Contudo, a quantidade de dados semiestruturados e não estruturados que precisam ser analisados é cada vez maior, o que causa um aumento dessa desvantagem.



Sendo assim, o Hive inverte a lógica e faz uso do conceito contrário: *scheme on read*. A estrutura dos dados, também conhecida como *esquema*, só é definida durante a leitura dos dados. Isso permite que os dados sejam armazenados do modo como são recebidos, estruturados ou não. Assim, os dados são estruturados apenas durante o processo de leitura de maneira muito mais específica para a forma como ele vai ser utilizado. Por essa característica, podemos dizer que essa estratégia é adequada para a utilização dos dados contidos no HDFS, uma vez que o sistema de arquivos distribuído não exige que os dados armazenados sejam estruturados. Por isso, o Hive consegue simular a utilização de um banco de dados que se aproveita das características de um sistema de arquivos distribuídos, como é o HDFS.

Os dados armazenados pelo Hive podem ser particionados em subdiretórios. Isso permite uma grande otimização, uma vez que limita a quantidade de dados em que a consulta ocorre. O Hive mantém em um banco de dados relacional (Derby ou MySQL) os metadados necessários para suportar os esquemas que definem como os dados devem ser lidos e os particionamentos que definem como eles devem ser organizados no sistema.

1.2 HiveQL

HiveQL é uma linguagem para consultas implementada pelo Hive. Como todos os dialetos SQL, não é totalmente compatível com uma revisão particular do padrão ANSI SQL. É muito parecido com MySQL, levando algumas pequenas diferenças. HiveQL não implementa os comandos *insert*, *update* e *delete* em linha, além de não implementar transações.

Hive implementa algumas extensões como as que oferecem melhorias de desempenho no contexto do Hadoop e as que servem para integrar extensões customizadas ou programas externos.

1.2.1 Database

O conceito de *database* em Hive consiste em um catálogo (*namespaces*) de tabelas. Essa definição é muito útil para evitar nomes de tabelas duplicadas. Se nenhuma database for especificada, uma database *default* é utilizada por padrão. A maneira mais simples de criar uma database é da seguinte forma:

```
hive> CREATE DATABASE nova_database;
```



Assim, se *nova_database* já existir, o Hive retornará um erro. Para evitá-lo, é possível utilizar a cláusula IF NOT EXISTS para não criar uma database que já existe. Isso é muito interessante para ser utilizado em *scripts* que precisam criar databases durante a execução:

```
hive> CREATE DATABASE IF NOT EXISTS nova_database;
```

O uso da palavra reservada SCHEMA pode substituir DATABASE em todos os comandos relacionados à database. A qualquer momento se pode verificar as databases existentes com a operação SHOW:

```
hive> SHOW DATABASES;
default
nova_tabela
```

Pode-se usar expressões regulares para filtrar as databases desejadas, utilizando as palavras reservadas LIKE ou RLIKE da seguinte forma:

```
hive> SHOW DATABASES LIKE 'n.*';
nova database
```

Para cada database, é criado um diretório. As tabelas de cada database são armazenadas em subdiretórios do diretório da database. A exceção, porém, são as tabelas da database default que não têm um diretório próprio. O diretório da database é criado no diretório especificado pela propriedade *hive.metastore.warehouse.dir*. Por padrão, o diretório especificado é o /user/hive/warehouse. Ao criar a database *nova_database*, o Hive irá criar o diretório /user/hive/warehouse/nova_database.db. É possível forçar a localização do novo diretório utilizando a palavra reservada LOCATION.

Para remover a database, utiliza-se a operação DROP:

```
hive> DROP DATABASE IF EXISTS nova_database;
```

O IF EXIST é opcional e, assim como no CREATE serve para evitar erros, caso a database não exista. Por padrão, o Hive não permite que uma database com contenha tabelas seja removida. Dessa forma, é necessário remover as tabelas primeiro, ou utilizar a palavra reservada CASCADE, que faz isso automaticamente.



1.2.2 Criação de Tabelas

Em Hive, a declaração CREATE TABLE segue os padrões de SQL estabelecidos. Mas, além disso, o Hive estende o suporte à flexibilidade em relação aos lugares em que os dados podem ser armazenados, os formatos de dados, entre outras extensões. É possível definir uma database prefixando seu nome antes do nome da tabela, separados por um ponto:

```
hive> CREATE TABLE IF NOT EXISTS mydb.funcionarios;
```

Da mesma forma, podemos utilizar a expressão IF NOT EXISTS para suprimir os erros no caso de já existir uma tabela com esse nome. No entanto, dessa forma, o Hive não o avisará se o esquema da tabela existente for diferente da tabela que está se tentando criar. Caso você deseje criar um esquema para essa tabela, você pode removê-la com DROP e recriá-la. No entanto, todos os dados serão perdidos. Dependendo da situação, pode ser possível executar algumas operações de ALTER TABLE para modificar o esquema de uma tabela existente. Também é possível utilizar a expressão LIKE para copiar o esquema de uma tabela previamente existente.

Para popular tabelas com dados, podemos utilizar a operação LOAD DATA e mover os dados de um sistema distribuído para o Hive. Na prática, o comando apenas indica que os dados serão utilizados pelo Hive, e os dados serão movidos para onde precisam estar. E podemos utilizar a operação LOAD DATA LOCAL para copiar os dados do sistema de arquivos local para o Hive. Dessa forma, as operações efetuadas nas tabelas não se refletem nos dados originais. É possível obter o mesmo efeito utilizando a expressão CREATE EXTERNAL para criar tabelas sem vínculo com os dados de onde elas foram importadas.

Para partitionar os dados de uma tabela, utilizamos a expressão PARTITIONED BY. Dessa forma, os dados armazenados são divididos por uma propriedade específica em subdiretórios:

```
hive> CREATE TABLE funcionarios(
        nome STRING
        setor STRING
    ) PARTITIONED BY (setor STRING)
```



TEMA 2 – INTEGRANDO HADOOP COM BANCO DE DADOS RELACIONAIS

Hive é uma ferramenta poderosa para visualizar dados inseridos no sistema de arquivos distribuídos do Hadoop como se eles estivessem em um banco de dados relacional. No entanto, muitas vezes, os dados que desejamos incluir em nossa análise já estão armazenados em um banco de dados externos ao Hadoop, como MySQL, Postgress e outros. Bancos de dados relacionais, como o MySQL, geralmente têm uma estrutura de armazenamento monolítica, ou seja, os dados estão instalados localmente em apenas um servidor, ao contrário do armazenamento feito pelo HDFS no Hadoop. Por essa característica, são ferramentas mais adequadas à OLTP (*OnLine Transaction Processing*), sistemas que registram todas as transações de determinada operação. São muito utilizados em sistemas financeiros e bancários, pois as transações devem ser atômicas e consistentes. Além disso, tais sistemas, na maioria das vezes, necessitam que o processamento das consultas seja muito rápido, de forma a manter a integridade dos dados em um ambiente com acessos variados e concorrentes. No entanto, devido a sua natureza distribuída, o armazenamento baseado no HDFS tem consultas mais complexas e lentas.

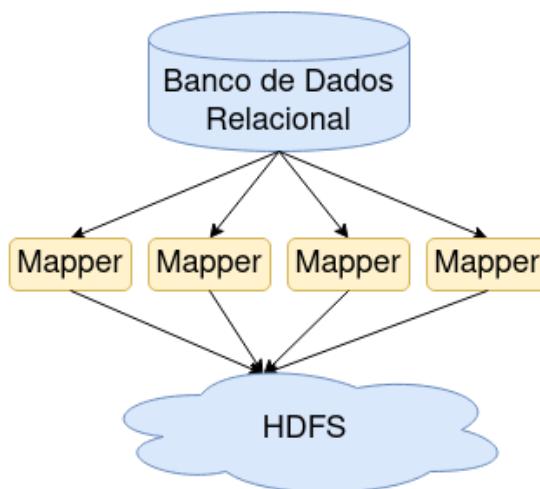
Dessa forma, existem muitos cenários em que dados que operam em bancos de dados relacionais possam ser analisados por si só, ou em conjunto com outros dados. Tornando evidente a necessidade de ferramentas que transferem dados entre fontes de dados estruturados, semiestruturados e não estruturados.

2.1 Importando dados com Sqoop

Sqoop é uma ferramenta projetada para transferir grandes volumes de dados estruturados, semiestruturados ou não estruturados de maneira eficiente entre o Hadoop e aplicações externas. Para isso, ele utiliza o MapReduce ou o Hive para fornecer operações paralelas e tolerantes a falhas. A maioria dos aspectos pode ser customizada. É possível controlar as linhas e colunas específicas a serem importadas; determinar delimitadores, caracteres de escape para as representações de dados baseados em arquivos, assim como o formato dos dados; a quantidade de tarefas MapReduce, entre muitas outras características.



Figura 1 – Fluxo de importação de dados do Sqoop



Fonte: Luis Henrique Alves Lourenço.

Para realizar a importação de dados por meio de um banco de dados MySQL, podemos utilizar o seguinte comando:

```
sqoop import --connect jdbc:mysql://localhost/foobar -m1 --  
driver com.mysql.jdbc.Driver --table foo
```

Assim, podemos utilizar o conector JDBC (*Java DataBase Connector*) como driver que conecta o banco de dados ao Sqoop. O parâmetro `-m` é utilizado para definir a quantidade de Mappers que serão criados para executar a operação. E o parâmetro `--table` é utilizado para especificar a tabela que será importada. O Sqoop também oferece formas de especificar o nome do arquivo criado no HDFS.

Primeiramente, o Sqoop analisa a base de dados e cria os metadados que serão utilizados para recuperar os dados. Em seguida, ele requisita trabalhos de MapReduce para transferir os dados. Cada linha do arquivo criado pelo HDFS representa uma entrada da tabela e valores separados por vírgula para cada coluna. É possível configurar qualquer caractere para fazer a separação dos valores. Além disso, é possível utilizar o parâmetro `--hive` para importar os dados do banco diretamente para o Hive.



2.2 Importação incremental

O Sqoop oferece um modo de importação incremental que pode ser utilizado para retornar apenas as linhas mais novas do que as importadas anteriormente. Dessa forma, é possível manter os dados do Hadoop sincronizados com o banco de dados relacional. Dois modos de importação incremental são suportados pelo Sqoop pelo parâmetro `--incremental`: `append` e `lastmodified`. O modo `append` deve ser utilizado quando dados são inseridos continuamente na tabela importada e uma das colunas armazena um valor que é incrementado. Tal coluna deve ser indicada pelo parâmetro `--check-column`. Assim, o Sqoop importa às linhas que a coluna verificada tem um valor maior do que o especificado pelo parâmetro `--last-value`. A estratégia de importação incremental definida pelo modo `lastmodified` foi projetada para ser utilizada quando os dados da tabela possam ser atualizados, e tal atualização escreve o `timestamp` atual na coluna indicada pelo parâmetro `--check-column`. As linhas com `timestamp` mais recente que o `timestamp` indicado pelo parâmetro `--last-modified` são importadas.

Ao final de uma operação de importação incremental, o valor que deve ser especificado como `--last-value` para a importação seguinte é retornado, de maneira que, ao realizar a importação seguinte, pode ser especificado como `--last-value` para garantir que apenas os dados novos ou atualizados sejam importados. Isso é tratado automaticamente ao criar uma importação incremental como um trabalho salvo (`saved job`). O `sqoop-job` é a ferramenta do Sqoop para armazenar os parâmetros de uma operação e poder reutilizá-la. Essa é a forma mais adequada de criar um mecanismo recorrente de importação incremental e, assim, manter os dados do Hadoop e do banco de dados sincronizados.

TEMA 3 – BANCO DE DADOS NÃO RELACIONAIS

Como vimos anteriormente, bancos de dados relacionais ainda hoje são adequados para muitas aplicações. Por isso, são uma fonte muito grande de dados valiosos. Graças a tecnologias como Hive e Sqoop, podemos integrá-los e fazer uso dos dados dessas aplicações para analisar em conjunto com uma grande quantidade de dados.

No entanto, como sabemos, bancos de dados relacionais e consultas SQL apresentam limitações, principalmente quando queremos analisar grandes



volumes de dados distribuídos em *clusters*. É para esse tipo de situação que se torna muito importante entendermos a tecnologia por trás dos bancos de dados não relacionais, que renunciam à linguagem de consulta SQL e são capazes de processar dados semiestruturados e não estruturados. Bancos de dados não relacionais, também conhecidos como NoSQL (*No SQL* ou, em alguns casos, *Not Only SQL*), são capazes de escalar horizontalmente de forma ilimitada.

Em NoSQL, utilizamos o conceito de teorema CAP para descrever o comportamento de um sistema distribuído. Esse teorema ajuda a explicar a relação que cada estratégia de banco de dados NoSQL adota quando eles têm a necessidade de requisitar uma operação de escrita seguida de uma operação de leitura, de forma que não necessariamente tais operações vão utilizar os mesmos nós do cluster. Assim, o teorema prevê que um sistema distribuído pode garantir apenas dois de três comportamentos. São eles: consistência, disponibilidade (*availability*) e tolerância a falhas ou tolerância a partição (*partition tolerance*). A consistência garante que o sistema sempre vai ler o dado mais atualizado, ou seja, tão logo ocorra a escrita de um dado no sistema, este dado pode ser lido por meio de qualquer nó do sistema. A disponibilidade implica que o cliente deve ter acesso aos dados mesmo na ocorrência de falhas. E a tolerância a falhas garante que mesmo que partes do sistema se encontrem desconectados por uma falha na rede, o sistema continua operando normalmente.

3.1 NoSQL

Os volumes de dados em muitos casos podem ser tão grandes que os bancos de dados relacionais não suportam escalar mais, pois dependem majoritariamente da escalabilidade vertical. Como já vimos, a escalabilidade vertical não tem a capacidade de escalar de forma ilimitada, pois depende dos avanços tecnológicos. Mesmo fazendo uso de diversas técnicas, como denormalização dos dados, que aumenta a redundância do banco, mas pode causar riscos à integridade dos dados; o uso de múltiplas camadas de cache; visões materializadas (*materialized views*), que é um objeto com os resultados de uma consulta; a utilização de uma configuração mestre-escravo que permite a replicação de um banco; ou a simples divisão dos dados em diferentes bancos. Todas essas técnicas otimizam a forma com que os bancos de dados relacionais lidam com grandes volumes de dados. Mas, ao mesmo tempo, seu uso pode



deixar o sistema excessivamente complexo. Podemos dizer, então, que para um volume muito grande de dados, a escalabilidade vertical dos bancos de dados relacionais pode não ser o suficiente.

Uma das principais características dos bancos de dados não relacionais, ou NoSQL, é a de não utilizar o modelo relacional. O modelo relacional se baseia no princípio de que os dados estão em tabelas, na lógica de predicados e na teoria de conjuntos. Os bancos de dados não relacionais, por outro lado, permitem que os dados sejam acessados de forma não relacionais, ou seja, sem precisar obedecer às regras da lógica de predicados e da teoria de conjuntos. Dessa forma, os modelos de bancos de dados não relacionais não são capazes de garantir transações com propriedades ACID. No entanto, não há um modelo único. Os diversos modelos de bancos de dados não relacionais são classificados em:

- **Bancos de dados de esquema chave-valor** – são bancos que utilizam um método de chave-valor para armazenar os dados também conhecido como *array associativo*. Os dados são armazenados em um conjunto de pares chave-valor em que a chave é um identificador exclusivo que permite a busca pelos valores. Em muitas implementações, a chave e os valores podem ser qualquer tipo de dado. Tanto a chave quanto os valores podem ser texto, números, estruturas de dados complexas ou até mesmo objetos binários. No entanto, chaves muito grandes podem ser ineficientes. Além disso, as chaves têm um modelo discretamente ordenado que as mantêm em ordenação lexicográfica. Essa característica é muito útil quando os dados estão distribuídos pelo *cluster*. É o modelo NoSQL mais simples. Ele permite rápido acesso aos dados pela chave por meio de uma API simplificada.
- **Bancos de dados baseados em documentos** – são bancos que utilizam o conceito de documentos para armazenar os dados. Cada implementação pode divergir a respeito de como os documentos são definidos, porém, em geral, todos assumem que documentos devem ser capazes de encapsular e codificar os dados. Para isso, formatos como XML, YAML, JSON e BSON são comuns. Há um esquema altamente flexível e com solução muito adequada para dados semiestruturados. Cada documento é endereçado por uma chave única, de maneira muito parecida com os bancos de esquema chave-valor. Além disso, ele oferece



uma API capaz de consultar documentos baseado em seus conteúdos, e tais bancos normalmente utilizam uma terminologia diferente. Comparado com os bancos de dados relacionais, podemos dizer que coleções são análogas a tabelas, documentos são registros (ou linhas), e as colunas são campos. Os campos são bastante diferentes neste caso, pois cada documento em uma coleção pode apresentar campos completamente diferentes.

- **Bancos de dados baseado em colunas** – são bancos de dados que utilizam o conceito de famílias de colunas para armazenar os dados em registros de tabelas. Uma família de colunas consiste em um conjunto de colunas opcionais. Dessa forma, cada registro não precisa dispor de dados nas mesmas colunas que os demais. É uma estratégia muito útil para dados esparsos. Cada valor é indexado por linha, coluna e *timestamp*.
- **Bancos de dados baseados em grafos** – são bancos que utilizam os conceitos desenvolvidos na teoria dos grafos. Um grafo é composto de dois elementos: um nó e uma relação. Cada nó representa uma entidade (que é parte do dado, como uma pessoa, um lugar, uma coisa, uma categoria etc.). E cada relação é a representação de como dois nós estão relacionados. Neste modelo de bancos, a relação entre os dados é normalmente mais relevante do que os dados em si. Dessa forma, é importante que as relações entre os dados sejam persistentes durante todo o ciclo de vida dos dados. Alguns bancos de dados baseados em grafos utilizam um tipo de armazenamento especificamente projetado para estruturas de grafos. No entanto, outras tecnologias podem utilizar bancos relacionais, baseados em documentos ou colunas, como camada de armazenamento em uma estrutura lógica de grafos. Além disso, os bancos de dados baseados em grafos implementam um modelo de processamento, chamado de adjacência livre de índices. Com isso, dados relacionados entre si apontam fisicamente um para o outro no banco de dados.

3.2 HBase

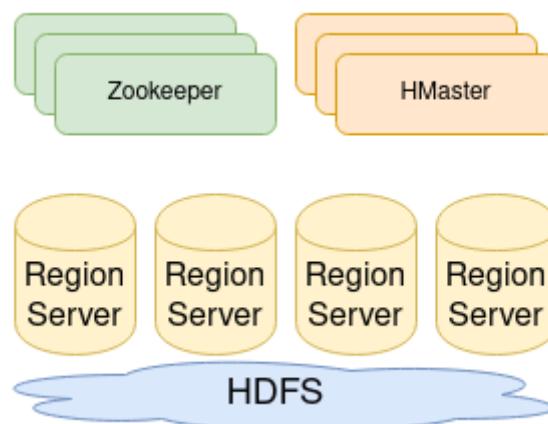
O HBase é um banco de dados não relacional projetado para utilizar o sistema de arquivos distribuído Hadoop (HDFS) da mesma forma que o BigTable



é utilizado sobre o Google File System. BigTable é um sistema de armazenamento distribuído para gerenciar dados estruturados, projetado para arquivos muito grandes (*petabytes* de dados distribuídos em milhares de computadores). Dessa forma, o HBase é o seu equivalente *open source* projetado para o Hadoop. O HBase não utiliza uma linguagem de consulta como o SQL. No entanto, utiliza uma API Java baseada no padrão CRUD. Sua arquitetura permite garantir acessos de escrita e leitura fortemente consistentes, em vez de eventualmente consistentes, ou seja, um dado inserido pode ser lido de forma correta instantaneamente de qualquer parte do *cluster*. Além disso, o HBase suporta processamento massivamente paralelizado por meio do MapReduce.

Tabelas são distribuídas pelo *cluster* por um conceito conhecido por *Regiões (regions)*. Regiões são o elemento básico da disponibilidade e distribuição das tabelas, e são gerenciados pelos *RegionServers*, executados nos *DataNodes*. É o componente responsável pelo particionamento automático dos dados de cada tabela pelo *cluster* que podem ser divididos e redistribuídos à medida que os dados crescem.

Figura 2 – Arquitetura do HBase



Fonte: Luis Henrique Alves Lourenço.

3.2.1 Arquitetura do HBase

O cliente HBase consulta a lista de todas as regiões no sistema, que é armazenada pelo Zookeeper, para localizar o RegionServer que oferece a parcela de registros em que se encontra o dado de interesse. Após localizar o RegionServer, o cliente se comunica com ele para requisitar as leituras e escritas



desejadas. Internamente, o cliente HBase utiliza um registro de conexões para obter o endereço do servidor mestre ativo, localização das regiões, e a identificação do *cluster*.

HMaster é a implementação do servidor mestre. O servidor mestre é responsável por monitorar todos os RegionServers do *cluster* e opera como interface para todas as alterações de metadados, como esquemas e particionamentos. Em um *cluster*, ele normalmente executa em um *NameNode*. O HBase pode ser configurado para executar mais de um servidor mestre. Dessa forma, um deles deve ser eleito como ativo. Esse gerenciamento ocorre por meio do Zookeeper, como já vimos.

3.2.2 Modelo de dados

Uma linha (*row*), ou registro, no HBase, é composta por uma chave e uma ou mais colunas com valores associadas a elas. As linhas são ordenadas alfabeticamente por suas chaves à medida que são armazenadas. Por esse motivo, a modelagem da chave de cada linha é muito importante, pois é muito relevante para manter dados relacionados em regiões próximas.

Colunas são agrupadas em conjuntos de famílias de colunas. Uma coluna consiste em uma família de colunas e um qualificador, normalmente utilizado signo dois pontos (:). Cada linha tem as mesmas famílias de colunas, mas não precisa que as mesmas colunas sejam preenchidas para todas as linhas. O qualificador é o que determina cada uma das colunas em uma família de colunas. Por exemplo, se temos uma família de colunas chamada *conteúdo*, podemos ter como qualificadores de coluna *conteúdo:html* ou, ainda, *conteúdo:pdf*.

Célula é a combinação de linha, coluna e *timestamp*. Permite que cada valor armazenado em HBase tenha versões.

Esses conceitos são expostos pela API do HBase ao cliente. A API para manipulação de dados consiste em três operações principais: *Get*, *Put* e *Scan*. *Get* e *Put* são específicos para cada linha e necessitam de uma chave. *Scan* é executado sobre um conjunto de linhas. Tal conjunto pode ser determinado por uma chave de início e uma chave de parada, ou pode ser a tabela toda sem chave inicial ou chave de parada. Muitas vezes, é mais fácil compreender o modelo de dados do HBase como um *map* multidimensional.



TEMA 4 – BANCO DE DADOS NOSQL EXTERNOS

Como vimos, os bancos de dados NoSQL têm diversas aplicações. Muitos deles são construídos por meio de estruturas que não incluem o uso de HDFS. Mesmo assim, em muitos casos, pode ser interessante incluir os dados de tais sistemas para serem analisados por meio das ferramentas disponibilizadas pelo ecossistema Hadoop. Neste tema, conheceremos alguns bancos NoSQL que não são projetados utilizando o HDFS, mas que são muito interessantes de serem integrados em sistemas Hadoop.

4.1 Cassandra

Cassandra é um banco de dados NoSQL distribuído e *open source* baseado no modelo de armazenamento chave-valor. Apesar de ser um banco de dados com características NoSQL, ele se diferencia dos demais ao implementar uma linguagem de consultas semelhante ao SQL conhecida por CQL. Os dados são armazenados em tabelas, linhas e colunas. Foi projetado para priorizar confiabilidade, escalabilidade e alta disponibilidade. No que se refere ao teorema CAP, Cassandra junta aos bancos que priorizam a disponibilidade e a tolerância falhas, renunciando à consistência, ou seja, podemos dizer que é um banco de consistência eventual. Cassandra oferece ainda capacidade de replicar dados por todos os nós do *cluster* (importante lembrar que Cassandra não utiliza um *cluster* Hadoop), garantindo a durabilidade e confiabilidade dos dados.

Cassandra suporta transações leves que suportam parcialmente as propriedades de transação ACID. Outra diferença de Cassandra para os bancos relacionais é a falta de suporte a operações JOIN, chaves estrangeiras, e não tem o conceito de integridade referencial, uma vez que se trata efetivamente de um banco de dados NoSQL.

A definição de dados em Cassandra determina que o objeto de nível mais alto é o *keyspace*, que contém tabelas e outros objetos como visões materializadas (*materialized views*), tipos definidos por usuário, funções e agregados. A replicação de dados é gerenciada em nível de *keyspace*.

O modelo de dados implementado em Cassandra foi projetado para ser um modelo de domínio simples e fácil de entender do ponto de vista de bancos relacionais, mapeando tais modelos para um modelo distribuído de tabelas de



dispersão (*hashtable*). No entanto, existem grandes diferenças entre Cassandra e bancos de dados relacionais. Cassandra não tem suporte à operação JOIN, de forma que se for necessário realizar uma operação de JOIN, você terá que fazer isso em um cliente externo, ou denormalizar os dados em uma segunda tabela. Não existe o conceito de integridade referencial em Cassandra. Ao contrário dos bancos de dados relacionais, que precisam trabalhar com dados normalizados, Cassandra opera de forma mais eficiente com dados denormalizados, ou seja, livres de esquemas. Por isso, quando se necessita realizar operações de JOIN, o recomendado é que os dados sejam denormalizados em uma segunda tabela. Essa abordagem facilita a distribuição dos dados pelo *cluster*. Diferentemente dos bancos relacionais que utilizam a estratégia de *scheme on write*, Cassandra, assim como outros bancos NoSQL, utiliza a estratégia de *scheme on read*, pois, como vimos, é muito mais eficiente armazenar dados denormalizados em bancos distribuídos.

4.1.1 Protocolo Gossip

Cassandra utiliza uma arquitetura de *cluster* descentralizado, ou seja, que não depende de um nó mestre para gerenciar os demais, evitando, dessa forma, que exista um único ponto crítico que possa afetar a disponibilidade do sistema. Cassandra implementa o protocolo Gossip para a comunicação entre os nós, descoberta de pares e propagação de metadados. Tal protocolo é responsável pela tolerância a falhas, eficiência e disseminação confiável de dados. Cada nó pode disseminar metadados para os demais nós, tais como informações de pertencimento ao *cluster*, *heartbeat*, e ao estado do nó. Cada nó mantém informações sobre todos os nós.

O protocolo Gossip executa uma vez por segundo para cada nó trocando informações de até três outros nós no cluster. Uma vez que o sistema é descentralizado, não há nó coordenando os demais. Cada nó seleciona de forma independente de um a três nós para trocar informações. E é possível que algum nó selecionado esteja indisponível. A troca de mensagens no Gossip funciona de maneira muito parecida com um *three-way-handshake* do TCP. Importante destacar que o protocolo gera apenas uma quantidade linear de tráfego de rede, uma vez que cada nó se comunica com até três outros nós.



4.1.2 Integração com Hadoop

Cassandra tem uma integração nativa com componentes do Hadoop. Inicialmente, apenas o MapReduce apresentava suporte a buscar dados contidos em Cassandra. No entanto, a integração amadureceu significativamente e hoje suporta nativamente o Pig e o Hive. Inclusive, o Oozie pode ser configurado para realizar operações de dados em Cassandra. O Cassandra implementa a mesma interface que o HDFS para fornecer a entrada de dados localmente.

4.2 MongoDB

O MongoDB é um banco de dados NoSQL com suporte corporativo, mas que se mantém *open source* e bem documentado, e modelo de dados baseado em documentos. É um banco que, em relação ao teorema CAP, privilegia a consistência dos dados e a tolerância a falhas. E, portanto, é um banco que pode estar eventualmente indisponível. Seu modelo de dados baseado em documentos tem uma estrutura muito semelhante ao formato JSON. Em MongoDB, dizemos que coleções são equivalentes às tabelas dos bancos relacionais e documentos são como os registros (ou linhas). Uma de suas características é dispor de um esquema muito flexível. Diferentemente dos bancos relacionais, o MongoDB não exige que se defina um esquema antes de inserir dados em uma tabela. Assim, cada documento em uma coleção pode ter campos completamente diferentes entre si. E os tipos de dados podem ser diferentes entre documentos de uma coleção. Além disso, é possível alterar a estrutura de cada documento acrescentando campos novos, removendo campos existentes, ou modificando os tipos de dados contidos em um campo. No entanto, na prática, os documentos de uma coleção compartilham uma estrutura similar. E o MongoDB permite a definição de regras para validar esquemas.

Uma vez que temos um modelo de dados muito flexível, precisamos ficar atentos à modelagem dos dados. Dessa forma, é muito importante definir como os dados serão estruturados e como serão representados os relacionamentos entre os dados. Para isso, temos modelos de dados embutidos ou referenciados.

Os modelos de dados embutidos, também conhecidos como *denormalizados*, armazenam pedaços de informação relacionada em um mesmo documento. Em geral, se utiliza quando as entidades apresentam uma relação



de pertencimento, ou uma relação de um-para-muitos entre si. Como resultado, é necessário menos consultas para realizar operações comuns. Dessa forma, obtém-se melhor desempenho de operações de leitura do banco, assim como a possibilidade de se obter dados relacionados com apenas uma operação. Ainda, permite que sejam atualizados dados relacionados com apenas uma operação de escrita. Em muitos casos, o uso de modelos de dados embutidos tem o melhor desempenho.

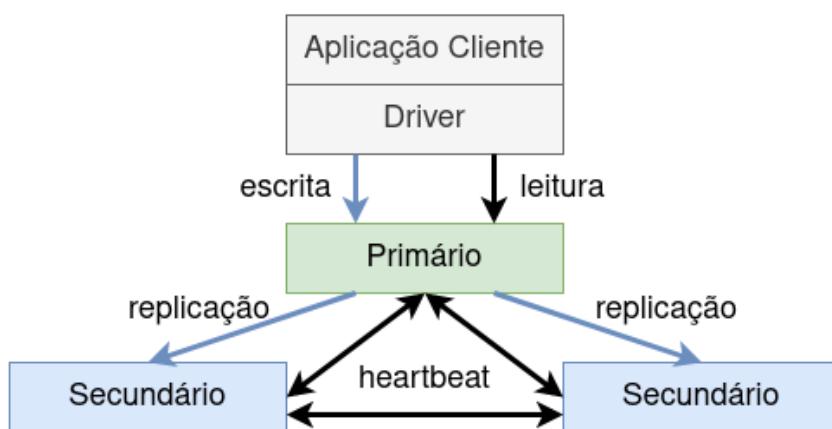
A alternativa aos modelos de dados embutidos são os modelos de dados normalizados. Tais modelos descrevem o relacionamento entre entidades utilizando referências entre documentos. Em geral, são utilizados quando embutir os dados gera uma duplicação de dados ineficiente, em que a eficiência de leitura não é suficiente para superar as implicações causadas pela duplicação; para representar relacionamentos muitos-para-muitos complexos; ou para modelar grandes conjuntos de dados hierárquicos. O MongoDB oferece funções de agregação para unir dados em diferentes coleções por meio de referências.

4.2.1 Conjuntos de replicação

O MongoDB tem uma arquitetura baseada em um único nó mestre. Isso significa que um único nó é responsável pela entrada de dados no banco e que, se esse único nó ficar indisponível, o banco inteiro também estará. Para contornar essa situação, o MongoDB implementa um conjunto de nós para replicar o nó mestre e o substituir em caso de indisponibilidade. Em MongoDB, um conjunto de replicação (*replica set*) é um grupo de processos *mongod* que mantém o mesmo conjunto de dados com múltiplas cópias em diferentes servidores. Os conjuntos de replicação oferecem redundância e melhoram a disponibilidade do banco.



Figura 3 – Arquitetura de conjuntos de replicação



Fonte: Adaptado de MongoDB.

Um conjunto de réplicas contém vários nós que armazenam cópias dos dados e, no máximo, um nó árbitro. Apenas um dos nós que armazenam dados é escolhido como o nó primário, que vai receber a conexão da aplicação cliente, enquanto os demais nós permanecem como nós secundários, que apenas replicam os dados do nó primário. Para isso, os nós secundários aplicam as operações armazenadas no *log (oplog)* do nó primário em seu próprio conjunto de dados de maneira assíncrona. Quando o nó primário se torna indisponível por alguma falha, um novo nó primário é eleito entre os nós secundários. Enquanto ele não for eleito, escritas no banco são desabilitadas. Para eleger um novo nó primário, a maioria dos nós do conjunto de replicação deve concordar sobre quem é o novo nó primário. Se o quórum de nós votantes for menor que a metade do total de nós votantes, não é possível eleger um nó primário e o banco permanece em modo somente-leitura. O ideal é manter um número ímpar de nós, pois, em caso de partição de rede, se o conjunto total de nós for ímpar, apenas o subconjunto que tiver mais do que a metade dos nós poderá receber escritas no banco. Para os casos em que não é possível ter um número ímpar de nós, é possível utilizar um nó árbitro, que não armazena réplicas dos dados e não pode ser escolhido como nó primário. O nó árbitro apenas serve para votar na escolha de um novo nó primário. A eleição se dá da seguinte forma: primeiro são selecionados os nós com maior prioridade. Se mais de um nó for selecionado, então, é escolhido o nó que estiver mais atualizado em relação ao nó primário indisponível. Se o nó primário indisponível voltar a estar disponível, uma nova eleição não é realizada, e o nó volta como um nó secundário.



4.2.2 Particionamento (*sharding*)

Os nós secundários não aumentam a escalabilidade do banco. Apenas são utilizados para melhorar a disponibilidade do sistema em caso de indisponibilidade do nó primário. No entanto, é possível combinar múltiplos conjuntos de replicação para aumentar a escalabilidade horizontal do banco de forma distribuída. Um *cluster* particionado (*sharded cluster*) contém os seguintes componentes:

- Partição (*shard*) – cada partição tem um subconjunto dos dados particionados. Cada partição pode ser atribuída a um conjunto de replicação.
- *Mongos* – são instâncias que atuam como uma interface entre a aplicação cliente e o *cluster* particionado para indicar em qual partição encontram-se os dados que se desejam consultar.
- Servidores de configuração (*config servers*) – conjunto de replicação utilizado única e exclusivamente para armazenar metadados e definições de configuração do cluster.

O MongoDB pode ser particionado de maneira que os dados sejam divididos em diversas partições. Os dados podem ser particionados em *chunks* para serem armazenados nas partições, ou podem ser armazenados sem ser particionados em uma única partição. Os dados particionados devem ter uma chave de partição (*sharded key*) para que sejam distribuídos pelas partições. O MongoDB oferece duas estratégias para distribuir os *chunks* de dados pelas partições. Os *chunks* podem ser distribuídos por faixa de valor das chaves de partição, ou por um *hash* calculado com a chave de partição. Ambas as estratégias são utilizadas para balancear a carga de dados pelas partições.

Além disso, o MongoDB implementa um *framework* de agregação que permite executar operações de MapReduce e um sistema de arquivos distribuídos, o GridFS, para acessar os dados do banco. Assim, é possível utilizar o MongoDB como um substituto do Hadoop. O MongoDB também implementa o suporte à integração com Hadoop, Spark, diversas linguagens, além de um conector SQL para realizar consultas SQL, com algumas restrições.



TEMA 5 – MOTORES DE CONSULTAS SQL

Muitos dos bancos de dados NoSQL não implementam o padrão SQL completamente. No entanto, em muitos casos, é necessária a integração entre as ferramentas armazenamento de grandes volumes de dados implementadas pelos bancos não relacionais por aplicações que utilizam consultas SQL. Além disso, a capacidade de unificar consultas para utilizar diversos bancos diferentes pode ser útil em várias aplicações. Para atender a essas demandas, existem diferentes soluções.

5.1 Drill

Drill é um motor de consultas SQL distribuído *open source* e mantido pela Fundação Apache para a exploração de grandes volumes de dados, por meio da combinação de uma variedade de bancos de dados não relacionais e arquivos.

De forma semelhante a bancos como MongoDB e Elasticsearch, o Drill utiliza um modelo de dados em formato JSON que não exige definição de esquema. Ele automaticamente entende a estrutura dos dados. Tal modelo de dados permite a consulta de dados semiestruturados complexos localmente, sem a necessidade de transformar antes ou durante a execução da consulta. Para isso, o Drill fornece extensões ao SQL para realizar consultas em dados aninhados.

Drill suporta a sintaxe do padrão SQL:2003. Isso significa que o Drill suporta diversos tipos de dados, incluindo DATE, INTERVAL, TIMESTAMP e VARCHAR, assim como permite a construção de consultas complexas, como subconsultas correlacionadas e JOIN em cláusulas WHERE. Dessa forma, o Drill é capaz de operar com ferramentas de BI, tais como Tableau, MicroStrategy, QlikView e Excel. É válido lembrar, porém, que operações muito complexas, especialmente envolvendo JOIN, podem não ser muito eficientes.

Uma característica muito importante do Drill é que, além de permitir consultas SQL, isso pode ser feito pelo uso de diversas fontes de dados, como Hive, HBase, MongoDB, sistemas de arquivos (local ou distribuído, como HDFS e S3). Com isso, é possível realizar um JOIN entre uma tabela Hive e uma HBase ou um diretório de arquivos de log. O Drill esconde toda a complexidade de realizar tais operações SQL em diferentes bancos de dados NoSQL e sistemas de arquivos como se fosse um banco SQL.



O Drill também oferece uma API Java para definir funções customizadas, permitindo que o usuário adicione sua lógica de negócios. Além disso, as funções customizadas para Hive também podem ser utilizadas em Drill.

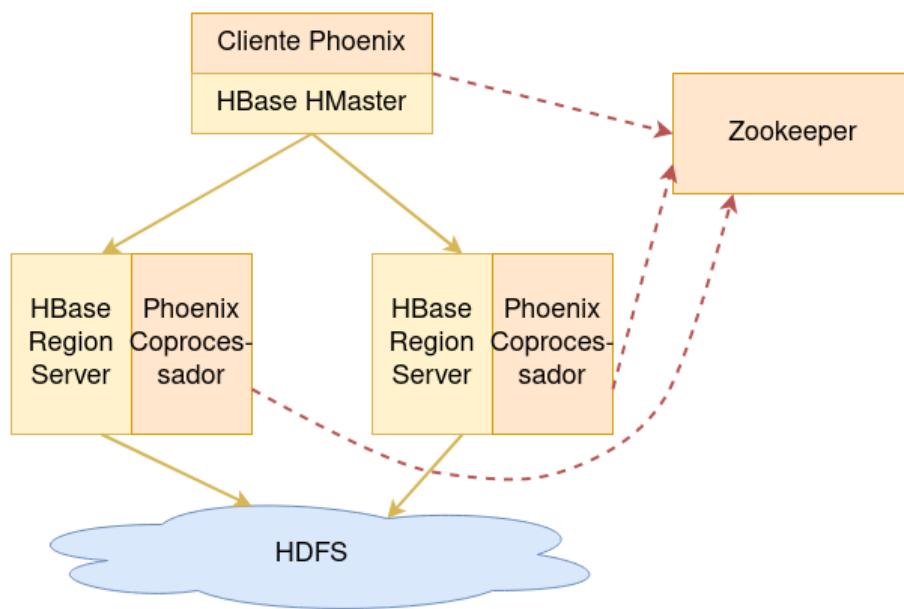
Para processar dados em larga escala, o Drill tem um ambiente de execução distribuída coordenado por um serviço chamado *Drillbit*, responsável por aceitar as requisições dos clientes, processar as consultas e retornar os resultados aos clientes. Ele pode ser instalado em todos os nós de um *cluster* Hadoop para formar um ambiente distribuído em cluster. Dessa forma, o Drill optimiza a execução das consultas por não precisar transferir dados entre os nós. Utiliza o Zookeeper para coordenar os serviços Drillbit em um *cluster*, apesar de poder executar em qualquer ambiente de cluster distribuído, não apenas o Hadoop. No entanto, sua única dependência é o Zookeeper.

5.2 Phoenix

Phoenix é um motor de banco de dados relacional *open source* projetado para operar sobre o HBase, podendo ser utilizado por aplicações que exigem baixa latência e que exigem a capacidade de operar transações OLTP. Isso permite ao Phoenix oferecer transações com propriedades ACID por meio de consultas SQL. O Phoenix traduz consultas SQL para uma série de operações HBase e gerencia a execução dessas operações para produzir um conjunto de resultados que podem ser obtidos por um conector JDBC ou outra interface. Além disso, o Phoenix suporta índices secundários, funções definidas por usuário (inclusive funções HBase), e é possível integrá-lo com MapReduce, Spark, Hive e Flume. Apesar de ser uma camada sobre o HBase, o Phoenix é muito eficiente, uma vez que é capaz de realizar otimizações em consultas complexas automaticamente.



Figura 4 – Arquitetura Phoenix



Fonte: Luis Henrique Alves Lourenço.

Uma aplicação pode se conectar a um cliente Phoenix via JDBC, por interface de linha de comando, por uma API Java. O cliente Phoenix analisa uma consulta SQL e a traduz em operações para o HBase. Como vimos alguns temas atrás, o HBase utiliza-se de servidores de região (*Region Servers*) para distribuir o processamento pelo *cluster*. Para cada *Region Server*, o Phoenix utiliza um componente com o nome de coprocessador para auxiliar na execução e otimização das operações planejadas pelo cliente Phoenix, que serão realizadas pelos *Region Servers*. Além disso, tanto o Phoenix quanto o HBase utilizam o Zookeeper para gerenciar quais *Region Servers* estão disponíveis.

5.3 Presto

Presto é um motor de consultas SQL distribuído *open source* para consultas analíticas interativas capaz de operar em volumes de dados muito grandes. Foi projetado para consultar dados nos locais onde eles estão armazenados, ou seja, interagindo com ferramentas como Hive, Cassandra, MongoDB, Kafka, JMX, PostgreSQL, Redis, Elasticsearch, bancos de dados relacionais, ou mesmo arquivos locais. Ele permite combinar dados de múltiplas fontes. É um motor otimizado para consultas analíticas e data warehouse (OLAP).



Presto não é um banco de dados. De fato, o que ele faz é entender requisições SQL e realizar as operações combinando os dados de diversas fontes de dados. Sua arquitetura é muito similar a um sistema gerenciador de bancos de dados utilizando computação em *cluster*. Pode ser compreendido como um nó coordenador trabalhando em sincronia com múltiplos nós trabalhadores. O Presto analisa operações SQL enviados pelas aplicações clientes e planeja a execução das tarefas dos nós trabalhadores. Os trabalhadores processam conjuntamente os registros das fontes de dados e produzem o resultado que é entregue à aplicação cliente. Para acessar as diversas fontes de dados, Presto implementa conectores específicos para cada ferramenta. Esquemas e referências de dados são armazenados em catálogos (*Catalogs*) e definidos em arquivos de propriedades no diretório de configuração do Presto. Os esquemas em conjunto com os catálogos são a forma utilizada para definir quais tabelas estão disponíveis para serem consultadas.

Presto executa operações SQL e converte tais definições em consultas que serão executadas por um *cluster* distribuído de um coordenador e seus trabalhadores. As operações SQL executadas pelo presto são compatíveis com o padrão ANSI.

FINALIZANDO

Nesta aula, nos aprofundamos nos temas relacionados ao armazenamento em bancos de dados. Começamos entendendo como funciona e para que serve o Hive e sua linguagem semelhante à SQL, HiveQL. Em seguida, aprendemos sobre o Sqoop e como fazer a integração de bancos de dados relacionais com o Hadoop.

Pudemos ver uma explicação sobre os bancos de dados NoSQL, também chamados de *bancos não relacionais*. Em seguida, conhecemos o banco NoSQL HBase, implementado sobre o HDFS, passando por bancos NoSQL que não utilizam a infraestrutura do Hadoop, como o Cassandra e o MongoDB.

Finalizamos conhecendo algumas tecnologias para realizar consultas SQL em bancos NoSQL. Primeiro, vimos como funciona o Drill, projeto da Fundação Apache que implementa um motor de consultas SQL que realiza operações conjuntamente em bancos de dados NoSQL e arquivos em sistemas de arquivos locais ou distribuídos. Conhecemos também o Phoenix, projeto da Fundação Apache que implementa um tradutor de consultas SQL para o HBase.



E, por último, vimos o Presto, um motor de consultas SQL que realiza consultas em diversos tipos de bancos relacionais, NoSQL e sistemas de arquivos.



REFERÊNCIAS

CAPRIOLI, E.; WAMPLER, D.; RUTHERGLEN, J. **Programming Hive.** Data Warehouses and Query Languages for Hadoop. Sebastopol CA: O'Reilly Media, inc., 2012.

JAIN, A. **Instant Apache Sqoop:** Transfer data efficiently between RDBMS and the Hadoop ecosystem using the robust Apache Sqoop. Birmingham UK: Packt Publishing Ltd., 2013.



BIG DATA

AULA 1

Prof. Luis Henrique Alves Lourenço



CONVERSA INICIAL

Nesta aula, será apresentado a você um panorama sobre o conceito de Big Data. Estudaremos o contexto do qual surge esse conceito, os fundamentos que o definem e que são importantes ao tema. E abordaremos as etapas necessárias para extrair informações valiosas dos dados e apresentá-las.

TEMA 1 – A ERA DOS DADOS

Os avanços tecnológicos das últimas décadas nos trouxeram cada vez mais capacidade para medir e avaliar os eventos que acontecem a cada instante à nossa volta. A necessidade de gerar informações valiosas com base nos dados obtidos por tantos mecanismos de medição tem sido discutida e estudada a fim de combinar o uso de tais tecnologias para capturar, administrar e processar a quantidade cada vez maior de dados gerados das mais distintas formas.

Desde a popularização dos computadores e demais equipamentos digitais, vivemos em uma época de transição entre um mundo em que todos os dados eram gerados e armazenados em mídias analógicas para outro, em que os dados são digitais. Com essa revolução digital, surge a possibilidade de um volume muito significativo de dados a explorar. Enquanto nos anos 1990 havia poucos setores digitalizáveis, limitados a alguns segmentos da música e mídia, no início dos anos 2000 setores como o comércio eletrônico e o internet banking iniciaram sua transição para o digital, e hoje quase todos os aspectos da vida cotidiana passam por formatos digitais.

Dados da web, mídias sociais, transações das mais diversas naturezas (entretenimento, financeiro, telecomunicações), dados biométricos, relatórios, logs, documentos e muitos outros tipos de dados gerados a cada instante permitem a construção de aplicações que antes pareciam impossíveis devido ao alto custo e complexidade. O volume e a variedade dos dados gerados continuam a crescer, tornando sua análise cada vez mais complexa. No entanto, é cada vez mais necessário que tais análises ocorram, para que se possa produzir informações valiosas em tempo real.

Para responder à demanda pelas informações valiosas que podem ser obtidas com os dados, deve-se prestar especial atenção a certos fatores para processá-los e analisá-los, como: a relevância dos dados; a velocidade



necessária para processá-los; quão variados precisam ser; qual seu nível de atualização; entre diversos outros fatores que podem ser cruciais para as informações extraídas dos dados e que podem refletir a realidade e gerar o valor esperado.

Dessa forma, começamos a definir o conceito de *Big Data* não só como uma solução empacotada que pode ser colocada em prática adquirindo certa tecnologia com um fornecedor, mas como o conjunto de práticas e técnicas que envolvem o processamento de um volume de dados confiáveis e variados com a velocidade necessária à geração de valor.

1.1 O crescimento do volume de dados

Um dos aspectos mais influentes no crescimento dos dados foi a conexão entre os equipamentos digitais pela internet. Em 1995, quando a internet estava nos primórdios, estima-se que menos de 1% dos dados eram armazenados em formato digital (Marquesone, 2016). A conexão entre diversos dispositivos eletrônicos permitiu a criação de uma diversidade antes inimaginável de serviços que hoje são amplamente utilizados, como a compra de passagens on-line, definição de trajeto por auxílio de GPS, reuniões por videoconferência, serviços de financiamento coletivo, busca e candidatura de vagas de trabalho on-line, serviços de streaming de vídeo e áudio, redes e mídias sociais, jogos on-line, compras via comércio eletrônico, compras coletivas, internet banking, entre tantos outros.

Outro fator importantíssimo foi a adoção em grande escala de dispositivos móveis, que só foram intensamente popularizados devido à redução do custo de produção de equipamentos com poder de armazenamento adequado. Mesmo que houvesse a intenção de explorar os dados gerados, enquanto não houvesse poder de processamento ou capacidade de armazenamento suficiente a custos acessíveis, a maioria dos dados seria simplesmente descartada. Portanto, o aumento no poder de processamento, combinado com a redução de custo de armazenamento, contribuiu com o aumento do volume de dados.

As empresas puderam explorar o potencial contido em diferentes tipos de dados. Dados obtidos por diversos tipos de sensores e equipamentos finalmente puderam ser analisados, passando a gerar valor, até o ponto atual, em que o recente desenvolvimento das novas tecnologias de redes móveis permite que



sensores cada vez menores possam gerar uma quantidade gigantesca de dados com a interação entre os próprios equipamentos ou seu ambiente, pelo conceito de *internet das coisas* (do inglês *internet of things* – IoT). Segundo Cesar Taurion (2013, p. 29),

A internet das coisas vai criar uma rede de centenas de bilhões de objetos identificáveis e que poderão interoperar uns com os outros e com os *data centers* e suas nuvens computacionais. A internet das coisas vai aglutinar o mundo digital com o mundo físico, permitindo que os objetos façam parte dos sistemas de informação. Com a internet das coisas podemos adicionar inteligência à estrutura física que molda nossa sociedade.

O crescimento da capacidade de processamento, segundo a lei de Moore, deve dobrar a cada 18 meses. Inicialmente esse crescimento ocorria devido à miniaturização dos processadores. Atualmente a paralelização do processamento permitiu que esse crescimento se mantivesse. Espera-se que no futuro o desenvolvimento de novas tecnologias, como a computação quântica, possa aumentar a capacidade de processamento. Além disso, as novas tecnologias de armazenamento, como os discos de estado sólido (SSDs), têm permitido seu barateamento e aumento de capacidade.

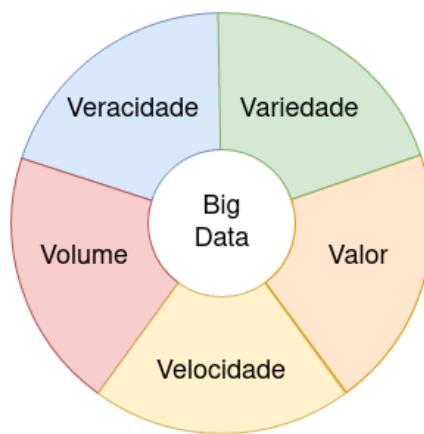
Podemos dizer que o aumento da geração de dados não deve ter seu ritmo reduzido tão cedo. Portanto, cada vez mais ferramentas poderosas são necessárias para analisar o imenso volume de dados gerados todos os dias no mundo.

TEMA 2 – OS Vs EM BIG DATA

Como acabamos de ver, o volume dos dados é um dos atributos mais relacionados ao conceito de Big Data, mas não é o único a defini-lo. Devemos considerar pelo menos outros dois aspectos: a velocidade em que os dados são processados e analisados, assim como a variedade dos dados, que podem ser obtidos de diversas fontes e se estruturar de diferentes formas.



Figura 1 – Os Vs em Big Data



2.1 Volume

O crescente volume de dados gerados a todo momento indica com relativa clareza que este é justamente o atributo mais significativo quando falamos em Big Data. No entanto, o que não fica muito claro é o ponto a partir do qual determinado conjunto de dados tem volume suficiente para ser considerado Big Data. Esse questionamento parte de uma premissa equivocada, pois o conceito de Big Data não pode ser definido única e exclusivamente pelo volume de dados processados. Um laboratório, por exemplo, pode necessitar de soluções de Big Data para visualizar imagens com 40 gigabytes, enquanto um observatório de astronomia pode necessitar de soluções de Big Data para analisar imagens e dados de sensores com terabytes de volume.

Portanto, o que define se um volume de dados necessita de soluções de Big Data não é seu tamanho, mas sua relação com a escalabilidade, eficiência, custo e complexidade. É o ponto em que a aplicação de soluções de Big Data supera os limites alcançados pelas tecnologias tradicionais que não foram projetadas para suportar esse volume.

2.2 Variedade

Um conjunto de dados de Big Data pode vir de diversas fontes e em diversos formatos. No entanto, as tecnologias tradicionais utilizam majoritariamente bancos de dados relacionais, ou seja, bancos que, embora



muito eficientes, são projetados para armazenar dados previamente estruturados, respeitando a propriedade Acid para que a integridade dos dados seja garantida da seguinte forma:

- **Atomicidade**: garante que transações não tenham atualizações parciais, ou seja, devem se comportar de forma indivisível e ser feitas por inteiro, ou então não são feitas;
- **Consistência**: garante que transações não afetem a consistência do banco. Dessa forma, as transações são completadas apenas se não ferirem nenhuma regra de integridade do banco, levando o banco de dados de um estado consistente a outro estado também consistente;
- **Isolamento**: garante que transações concorrentes não interfiram nos eventos umas das outras. As transações devem ter o mesmo resultado, como se fossem executadas uma após a outra;
- **Durabilidade**: garante que todos os efeitos de uma transação completada com sucesso persistam mesmo na ocorrência de falhas externas.

No entanto, estima-se que a maioria dos dados existentes seja de dados não estruturados ou semiestruturados (Marquesone, 2016). Os dados semiestruturados compreendem estruturas previamente definidas, mas que não exigem o mesmo rigor que os bancos de dados relacionais – é o caso de arquivos nos formatos JavaScript Object Notation (JSON) ou eXtensible Markup Language (XML). Já os dados não estruturados são todos aqueles excessivamente complexos para serem armazenados apenas com ferramentas tradicionais de armazenamento e gerenciamento de dados. É o caso de mídias como vídeos, imagens, áudios e até mesmo alguns formatos de texto.

Os dados não estruturados ou semiestruturados exigem que a tecnologia adotada forneça uma estrutura flexível o bastante para que dados tão diversos possam ser analisados. Além disso, a estrutura deve permitir a utilização de ambientes distribuídos, incluindo a variedade de soluções e tecnologias necessárias para atender a demanda específica que a solução requer.

Há uma variedade muito grande de dados em variados formatos, sendo utilizados por uma variedade também muito grande de soluções com necessidades muito específicas. A solução de Big Data deve ser capaz de integrar e interagir com toda essa diversidade de dados.



2.3 Velocidade

Devido à popularização da internet, das mídias e redes sociais, dos dispositivos móveis e da internet das coisas, dados são gerados de forma cada vez mais rápida e por cada vez mais agentes. Por exemplo, um único carro moderno pode ter cerca de 100 sensores, que geram dados a cada instante. Por isso a velocidade em que esses dados são coletados, analisados e utilizados se torna cada vez mais importante.

Dados perdem valor com o tempo; por exemplo, sites de comércio eletrônico que atualizam seus preços de acordo com a demanda de suprimentos podem maximizar as vendas. Outro exemplo são os serviços de tráfego, que podem oferecer melhores rotas ou sistemas críticos que dependem das informações geradas em tempo real. Portanto, para muitos serviços, de nada adianta ter a capacidade de processar um volume imenso de dados, de diferentes lugares e formas, se a solução não for capaz de responder no tempo necessário para que a informação seja útil e não perca seu valor.

2.4 Valor

Ao definir o conceito de Big Data, alguns autores fazem referência a apenas três fatores (Vs): **volume, variedade e velocidade**. Reunidos, definem Big Data como a coleta e análise de um volume imenso de dados que podem vir de diversas fontes e ter uma grande variedade de formatos, numa velocidade altíssima. No entanto, podemos avaliar também quão valioso e significativo um dado pode ser para uma solução. Dessa forma, temos como saber quais dados podem gerar maior valor para a solução e, por isso, devem ser priorizados. Ao priorizar ou escolher os dados corretos, é possível otimizar a solução para que o valor gerado seja o mais adequado.

2.5 Veracidade

Outro fator de grande importância é saber quão confiável é o conjunto de dados que estamos utilizando. Isso impacta diretamente na confiabilidade da informação extraída. Estima-se que dados de baixa qualidade custem à economia trilhões de dólares anualmente. Dados com uma precisão inadequada



ou falsos podem levar a informações incorretas e até inviabilizar soluções. Portanto, se a veracidade dos dados coletados não for avaliada e garantida, corremos o risco de afetar o valor e a validade das informações que geramos.

2.6 Temos uma definição para o conceito de Big Data?

Dado o que consideramos, a definição de Big Data pode variar bastante. Alguns autores podem resumir a definição nos três principais fatores (volume, variedade e velocidade), enquanto outros chegam a utilizar até dez fatores (ou mais). No entanto, não divergem muito dos 5 Vs que vimos até aqui. Muitas vezes acontece de alguns autores aglutinarem algumas ideias em um dos fatores ou as subdividem em novos fatores que podem ser mais adequados à solução específica que se está desenvolvendo.

Portanto, para nossos propósitos, vamos definir Big Data como o conjunto de práticas e técnicas que envolvem a coleta e análise de um grande volume de dados confiáveis e variados (tanto no formato quanto na origem), com a velocidade necessária para gerar o valor adequado à solução. O que realmente importa é o valor que podemos gerar quando nos livramos das limitações relativas ao volume, variedade, velocidade e veracidade dos dados processados.

TEMA 3 – OBTEÇÃO E ARMAZENAMENTO DE DADOS

No que diz respeito ao Big Data, tudo se inicia com a obtenção dos dados que serão processados. Como vimos, os dados podem vir de diferentes origens e ter diferentes formatos. Pode ser que os dados necessários ainda não existam e precisem ser gerados, que sejam internos à própria aplicação ou que devam ser buscados de fontes externas. Todas essas questões fazem parte da fase de obtenção de dados e, para isso, estratégias de como os dados serão coletados e armazenados devem ser definidas.

3.1 Obtenção de dados

A obtenção de dados pode ser compreendida com diferentes estratégias de captura e utilização de dados no projeto.



- **Dados internos:** são os dados que o proprietário do projeto (empresa) já tem e cujo controle já detém. Tais dados podem vir de sistemas de gerenciamento da própria empresa, como: sistemas de gerenciamento de projetos; automação de marketing; sistemas *customer relationship management* (CRM); sistemas *enterprise resource planning* (ERP); sistemas de gerenciamento de conteúdo; dados do departamento de recursos humanos; sistema do gerenciamento de talentos; procurações; dados da intranet e do portal da empresa; arquivos pertencentes à empresa, como documentos escaneados, formulários de seguros, correspondências, notas fiscais, entre outros; documentos gerados por colaboradores, como planilhas em XML, relatórios em PDF, dados em CSV e JSON, e-mails, documentos de texto em diversos formatos, apresentações e páginas web; e registros de log de eventos, de dados de servidores, logs de aplicações ou de auditoria, localização móvel, logs sobre o uso de aplicativos móveis e logs da web;
- **Dataficação:** é a transformação de ações sociais em dados quantificados de forma a permitir o monitoramento em tempo real e análises preditivas;
- **Dados de sensores:** são dados inseridos no contexto da internet das coisas, em que os objetos se comunicam com outros objetos e pessoas. Para esse tipo de solução, é necessário prover um meio de transmitir dados entre os sensores e um servidor capaz de armazenar os dados das interações. A obtenção de dados de sensores ocorre em tempo real, por isso os maiores desafios estão no volume e na velocidade com que os dados são gerados. Exemplos de dados de sensores são aqueles coletados de medidores inteligentes, sensores de carros, câmeras de vigilância, sensores do escritório, maquinários, aparelhos de ar-condicionado, caminhões e cargas;
- **Dados de fontes externas:** são dados obtidos de domínio público, como dados governamentais, dados econômicos, censo, finanças públicas, legislações, entre outros. Podemos considerar também qualquer tipo de dados obtidos por sites de terceiros, como mídias e textos de sites da web, além dos dados obtidos de mídias sociais. São basicamente todos os dados possíveis de obter por requisições na web ou uma *application programming interface* (API) dedicada. Muitos desses dados são



disponibilizados por APIs acessadas via *representational state transfer* (Rest), obtendo-se os dados requisitados em formato JSON.

3.2 Armazenamento

Como vimos, os bancos de dados relacionais foram por muito tempo o padrão mundial de armazenamento. Nesse modelo os dados são armazenados de forma previamente definida em estruturas de tabelas que podem ser relacionadas com outras tabelas da mesma base de dados. Uma das características mais importante desse tipo de armazenamento é o suporte a transações Acid.

Outra característica importante é o uso de *structured query language* (SQL) para operações de criação e manipulação de dados, o qual permitiu que os dados armazenados tivessem sua integridade garantida, e também permitiu gerar consultas mais complexas. No entanto, o crescimento constante na geração de dados mostrou os limites dos bancos de dados relacionais como única solução de armazenamento, principalmente no que se refere à **escalabilidade, disponibilidade e flexibilidade**.

3.2.1 Escalabilidade

Uma solução é considerada escalável quando mais carga é adicionada e mesmo assim o desempenho se mantém adequado. Com determinado volume de dados, os bancos de dados tradicionais conseguem manter esse desempenho apenas ao adicionar mais recursos computacionais à infraestrutura, o que é conhecido como *escalabilidade vertical*. No entanto, o volume de dados necessários aumentou tanto que esse tipo de solução não é mais viável em todos os casos, uma vez que os custos de tais recursos podem ser muito elevados.

3.2.2 Disponibilidade

Para que um serviço seja considerado de alta disponibilidade, o tempo em que ele se mantém operando deve ser priorizado em comparação às demais propriedades Acid. Portanto, deve-se garantir que o serviço se mantenha operando mesmo em casos de falha na infraestrutura.



3.2.3 Flexibilidade

Um serviço flexível é aquele capaz de comportar uma grande diversidade de dados. O grande problema desse requisito é que muitas vezes é inviável modelar um conjunto de dados de forma antecipada e que conte coleste características não estruturadas.

Concluímos que os bancos de dados tradicionais já não são a solução mais adequada para suprir os requisitos exigidos em soluções de Big Data. Dessa forma, soluções alternativas foram criadas para atender a esse tipo de demanda.

3.3 NoSQL

A noção de NoSQL incorpora uma ampla variedade de tecnologias de bancos de dados desenvolvidos como resposta à demanda de aplicações modernas. Quando comparadas com bancos de dados relacionais, bancos NoSQL são mais escaláveis, têm melhor desempenho, e seu modelo de dados resolve questões que os bancos de dados relacionais não foram projetados para resolver, como grandes volumes de dados de estruturados, semiestruturados e não estruturados que se modificam rapidamente, arquiteturas distribuídas geograficamente, entre outras. Os modelos de bancos de dados NoSQL podem ser classificados de acordo com a estrutura em que os dados são armazenados. Existem vários modelos, e os quatro principais são: **orientado a chave-valor**, **orientado a documentos**, **orientado a colunas** e **orientado a grafos**.

3.3.1 Bancos de dados orientados a chave-valor

Os bancos de dados orientados a chave-valor são os modelos mais simples de NoSQL. Cada item é armazenado como um atributo-chave normalmente composto de um campo tipo *string* associado a um valor que pode conter diferentes tipos de dados. Esse modelo não exige um esquema predefinido, como acontece nos bancos de dados relacionais. Esse tipo de banco de dados pode ser utilizado tanto para armazenar os dados quanto para mantê-los em *cache* para agilizar o acesso. Portanto, é um tipo de banco muito importante para aplicações que realizam muitos acessos aos dados. Apesar das



vantagens dos bancos de dados chave-valor, ele tem limitações. A única forma de realizar consultas é por meio da chave, uma vez que não é possível indexar utilizando o campo valor.

3.3.2 Bancos de dados orientados a documentos

Os bancos de dados orientados a documentos são uma extensão dos bancos de chave-valor, uma vez que também associam uma chave a um valor. Mas nesse caso o valor é necessariamente uma estrutura de dados chamada *documento*. A noção de documento é o conceito central desse tipo de banco de dados, e consiste em estruturas de um padrão definido, tal como XML, YAML, JSON, ou até formatos binários.

O documento pode se comportar de forma muito semelhante ao conceito de *objeto* em programação. Além disso, permite-se um conjunto de operações muito semelhantes ao padrão Crud: *creation* (inserção), *retrieval* (busca, leitura ou requisição), *update* (edição ou atualização), e *deletion* (remoção, deleção). Isso permite criar consultas e filtros sobre os valores armazenados, e não somente pelo campo-chave.

Outra característica desse banco é a alta disponibilidade, uma vez que permite trabalhar com a replicação de dados em *cluster*, garantindo que o dado ficará disponível mesmo em caso de falha no servidor.

3.3.3 Bancos de dados orientados a colunas

Os bancos de dados orientados a colunas são otimizados para buscas em grandes bancos de dados. Podem ser interpretados como bancos chave-valor bidimensionais; neles, o que seria uma tabela num banco de dados relacional seria um item identificado por uma chave associada a um valor que pode conter vários conjuntos de chave-valor.

Tais conjuntos são o equivalente ao campo de uma coluna de determinado item. Isso permite flexibilidade, tal que cada registro pode ter um número diferente de colunas. Os bancos orientados a colunas também podem ter o conceito de famílias de colunas. Cada família tem múltiplas colunas que são utilizadas em conjunto, de maneira semelhante às tabelas dos bancos de



dados relacionais. Dentro de uma família de colunas, os dados são armazenados linha por linha, de forma que as colunas de uma linha sejam armazenadas juntas.

Esse banco de dados é altamente adequado a soluções que necessitam trabalhar com volumes imensos de dados, alto desempenho, alta disponibilidade no acesso, armazenamento de dados e flexibilidade na inclusão de campos. Além disso, sua solução tolera eventuais inconsistências.

3.3.4 Bancos de dados orientados a grafos

Os bancos de dados orientados a grafos são muito úteis quando as relações entre os dados são mais importantes que os dados em si. Esse tipo de banco é utilizado para armazenar a informação sobre as redes de dados. Em vez de os dados serem formatados em linhas e colunas, são estruturados em vértices, arestas, propriedades para armazenar os dados coletados e os relacionamentos entre esses dados.

Os vértices representam entidades ou instâncias. Equivalem a um registro, uma linha dos bancos de dados relacionais, ou um documento num banco de dados orientado a documentos. As arestas (ou relações) são as linhas que conectam os vértices, representando a relação entre os dois vértices conectados. As arestas podem ser direcionais ou não direcionais, e propriedades são informações relacionadas com os vértices.

As soluções NoSQL não foram desenvolvidas para substituir os bancos de dados relacionais, mas para complementá-los. A tendência é adotar soluções híbridas, em que cada banco é utilizado onde possa ser mais adequado.

3.4 Governança de dados

Para qualquer empresa que adote estratégias de Big Data em suas soluções, é muito importante gerenciar dados de forma que seu uso seja o mais eficiente e confiável possível. A governança de dados inclui as pessoas, os processos e as tecnologias necessárias para proteger os ativos de dados da companhia, de forma a garantir que os dados da empresa sejam comprehensíveis, corretos, completos, confiáveis, seguros e detectáveis. De acordo com Marquesone (2016), os principais tópicos na governança de dados são:



- **Arquitetura dos dados:** é o que define o modelo para gerenciar ativos de dados, alinhando-se à estratégia da empresa para estabelecer requisitos e projetos de dados estratégicos que atendam a esses requisitos. Todas as políticas que padronizam os elementos de conjuntos de dados, os protocolos e boas práticas são criadas para garantir a adoção dos padrões definidos;
- **Auditoria:** comprehende que os dados devem permitir o próprio rastreio, e deve ser possível conhecer quando os dados foram criados, como estão sendo utilizados e quais os seus impactos;
- **Metadados:** são dados a respeito de outros dados. No contexto da governança de dados, é o que permite a visualização holística e açãoável de uma cadeia de suprimentos de informação. É o que permite gerenciar as alterações nos dados, a auditoria e rastreabilidade do fluxo de dados, e também a melhora na acessibilidade dos dados por buscas e mapas visuais;
- **Gerenciamento de dados-mestre (*master data management – MDM*):** dados-mestre são aqueles essenciais para o negócio de uma empresa. Podemos compreendê-los como o estabelecimento e gerenciamento de dados no nível organizacional, que fornecem dados-mestre precisos, consistentes e completos por toda a empresa e para parceiros de negócio;
- **Modelagem dos dados:** como vimos, é importante que toda a variedade de dados disponíveis seja modelada, de forma a permitir a utilização de padrões de dados, evitar redundâncias, definir como os dados serão utilizados, e encontrar as melhores formas de construir uma arquitetura de dados mais ágil e governável;
- **Qualidade dos dados:** comprehende os processos incorporados com o objetivo de aperfeiçoar a qualidade dos dados de forma que contenham menos erros, estejam mais completos e possam otimizar a utilidade dos dados. Inclui a criação de *profiles* de dados, estratégias de limpeza, filtragem e agrupamento de dados;
- **Segurança:** relaciona-se à gestão de risco relacionado à coleta, armazenamento, processamento e análise dos dados. Isso implica que todos os dados importantes e sensíveis devem ser utilizados de maneira correta e segura, de forma a prevenir o mau uso em todos os níveis. Pode-



se utilizar estratégias de criptografia, definição e proteção a dados sensíveis, políticas de proteção de integridade, disponibilidade, confiabilidade e autenticidade dos dados.

A governança de dados tem se tornado ainda mais importante com a adoção cada vez maior de soluções de Big Data, uma vez que a veracidade e o valor dos dados são diretamente influenciados por seus processos, permitindo a criação de modelos de negócios mais inovadores, confiáveis e eficientes.

TEMA 4 – PROCESSAMENTO DE DADOS

Tão logo os dados são capturados e armazenados, inicia-se a fase de processamento dos dados. Para isso, devemos avaliar algumas questões relacionadas ao processamento, como alocação de recursos, escalabilidade, disponibilidade, desempenho e o tipo de processamento.

4.1 Escalabilidade

Uma das questões mais importantes quando tratamos de um volume de dados que pode crescer imensamente é a escalabilidade. Um sistema escalável é aquele em que o desempenho não se deteriora com o aumento significativo de dados sendo processados. A capacidade de processamento da plataforma deve escalar proporcionalmente à demanda. Para isso, é necessário monitorar a execução de forma a impedir o esgotamento de recursos.

Outro aspecto importante é que não se deve sacrificar a disponibilidade da plataforma. Mesmo que ocorram falhas, o serviço deve manter-se ativo. No que se refere à escalabilidade, existem duas estratégias possíveis. Podemos adotar um modelo de **escalabilidade vertical** ou de **escalabilidade horizontal**.

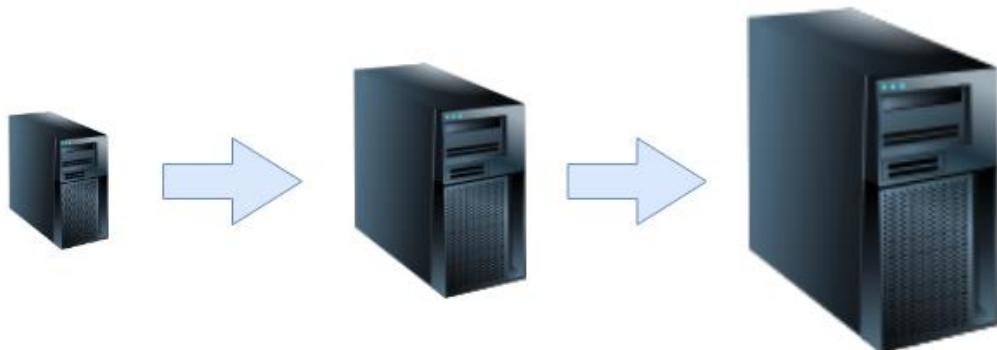
4.1.1 Escalabilidade vertical

A escalabilidade vertical se refere à adição de capacidade de processamento de um único recurso com a atualização da infraestrutura. Aumenta-se a capacidade de processamento da plataforma pela atualização da infraestrutura.



Esse tipo de estratégia costuma comprometer a disponibilidade do serviço, a não ser que haja redundância na infraestrutura. Sua vantagem é não exigir modificações nos algoritmos, mas em geral é uma solução que não atende à demanda quando aplicada ao contexto em que o volume de dados cresce rapidamente, como é o caso de soluções de Big Data.

Figura 2 – Escalabilidade vertical



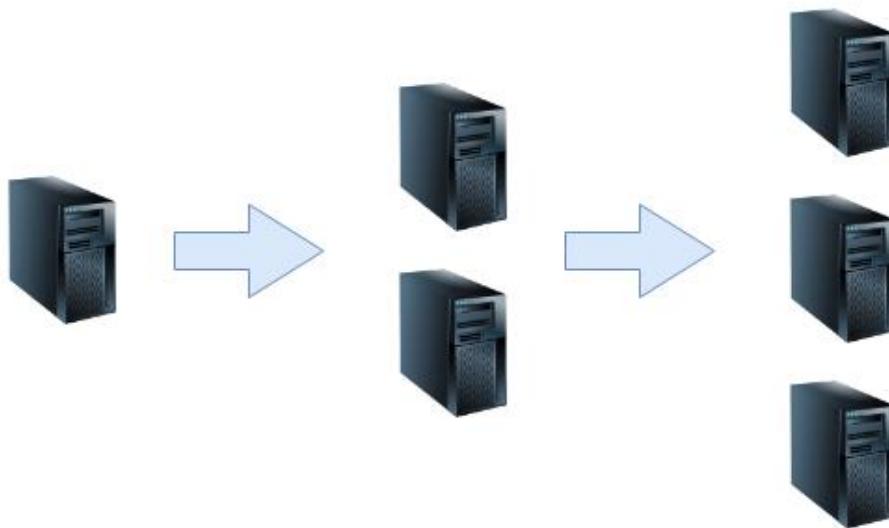
4.1.2 Escalabilidade horizontal

Uma estratégia muito mais adequada é a escalabilidade horizontal. Com ela o processamento é distribuído num conjunto de tarefas menores para serem processadas num *cluster* de recursos, de forma que o aumento da quantidade de recursos computacionais seja capaz de suprir o aumento na demanda de processamento. Além disso, os recursos do *cluster* se comportam de forma independente e redundante, colaborando com a disponibilidade da solução.

Dessa forma, o custo de melhorias na infraestrutura é reduzido, pois não é necessário interromper sua operação. A redistribuição de carga entre os recursos de um *cluster* é uma forma simples de regular a capacidade de processamento de acordo com a demanda; essa característica confere uma imensa capacidade de escalabilidade. Sua desvantagem é que a tecnologia tradicional não foi projetada para esse tipo de estratégia. Portanto, geralmente requer reimplementação de software para utilizar tecnologias de sistemas distribuídos.



Figura 3 – Escalabilidade horizontal



4.2 Processamento de dados com Hadoop

Originalmente, o Hadoop foi projetado para funcionar como um motor de busca de código aberto. O framework Hadoop se baseia em duas tecnologias que visam o suporte ao armazenamento e processamento distribuído de grandes volumes de dados:

- O sistema de arquivos distribuído Hadoop Distributed File System (HDFS);
- O modelo de programação distribuída MapReduce.

As principais características do Hadoop que envolvem o processamento de grandes volumes de dados são:

- **Baixo custo:** por ser projetado para utilizar em servidores tradicionais, não exige a implantação de hardware específico;
- **Escalabilidade:** devido à adoção de tecnologias distribuídas, sua capacidade de processamento escala linearmente. Isso significa que o aumento de recursos de computação se reflete diretamente na capacidade de processamento. E não é necessário alterar a base de código cada vez que a infraestrutura é atualizada;



- **Tolerância a falhas:** seu modelo de escalabilidade horizontal garante que, mesmo que algum dos recursos apresente falhas, os recursos restantes supram a demanda;
- **Balanceamento de carga:** a tecnologia de processamento distribuído evita gargalos que podem limitar o processamento de recursos. Dessa forma, todos os recursos operam de forma otimizada;
- **Comunicação entre máquinas e sua alocação:** ocorre de forma transparente para o usuário.

Todas essas características são implementadas pelo Hadoop e permitem que o desenvolvedor concentre seus esforços na lógica do problema. Dessa forma, a análise de um grande conjunto de dados – anteriormente ignorados devido a custos inviabilizantes – foi permitida com o surgimento das tecnologias distribuídas HDFS e Hadoop MapReduce.

O sistema de arquivos distribuídos HDFS foi criado para gerenciar o armazenamento das máquinas do *cluster*. Ele tem escalabilidade para armazenar grandes volumes de dados de forma tolerante a falhas, com recuperação automática. A disponibilidade é garantida pela replicação de dados, e o sistema se encarrega de quebrar o arquivo em blocos menores, replicando-os algumas vezes em diferentes servidores.

O Hadoop MapReduce foi projetado para gerenciar o processamento de dados distribuído com a divisão de uma aplicação em tarefas independentes executadas em paralelo nos servidores do *cluster*. O processamento é dividido nas seguintes etapas:

- **Map:** recebe uma entrada de dados e retorna um conjunto de pares no formato de pares chave-valor. As operações dessa etapa são definidas pelo desenvolvedor;
- **Shuffle:** os dados retornados pelo Map são organizados de forma a aglutinar todos os valores associados a uma única chave. Para cada chave teremos um par que a associa com uma lista contendo todos os valores relacionados a essa chave como valor. Essa etapa é feita automaticamente;
- **Reduce:** os dados organizados são recebidos, e operações definidas pelo desenvolvedor são realizadas, gerando o resultado da aplicação.



4.3 Processamento em tempo real

Apesar de todas as vantagens do Hadoop, ele não é adequado a todas as soluções de Big Data, uma vez que foi desenvolvido para processamento em lote. Isso significa que primeiramente são formados grupos de dados coletados num período de tempo, para só então os dados serem processados. Desde que os dados tenham sido gerados até seus resultados serem processados e, então, respondidos, temos uma quantidade de tempo significativa. Além disso, para muitos casos, o processamento dos dados deve se dar de forma contínua. No entanto, no modelo em lote, o processamento se encerra tão logo os resultados são retornados.

Diferentemente, muitas aplicações precisam que os dados sejam processados à medida que chegam à aplicação – ou seja, em tempo real –, e cada item de dados que chega à aplicação é processado imediatamente. Para isso, o processamento em tempo real tem alguns requisitos importantes:

- **Baixa Latência:** o tempo de processamento de um item de dado deve ser no máximo igual ao tempo em que novos dados chegam ao fluxo;
- **Consistência:** a solução deve ser capaz de operar com imperfeições e manipular inconsistências;
- **Alta disponibilidade:** etapas de coleta, transmissão e processamento de dados podem causar grandes impactos se ficarem indisponíveis, resultando na perda de dados significativos para a aplicação.

O processamento em tempo real é importante para soluções web com o rastreamento de usuários e análises de preferências, detecção de fraudes, redes sociais, com a identificação de tendências, além da internet das coisas, com milhares de objetos e sensores que geram dados o tempo todo.

Se soluções de processamento em lote, como o Hadoop, tiverem dificuldades em atender às demandas de velocidade necessárias para o processamento em tempo real, precisamos de uma solução que faça o processamento de fluxos de dados.



4.4 Processamento de dados com Spark

Spark é uma tecnologia que tem se destacado no processamento em tempo real, devido ao seu desempenho. Trata-se de um framework que estende o modelo de programação MapReduce, otimizando o desempenho em programação distribuída. O Hadoop compreende tanto um componente de armazenamento – o HDFS – quanto um componente de processamento – o Hadoop MapReduce. No entanto, o Spark concentra seus esforços no processamento de dados, podendo muitas vezes ser utilizado com o Hadoop, uma vez que seu processamento costuma ter desempenho muito superior ao Hadoop MapReduce.

Os principais componentes do Spark são:

- **Spark Core:** disponibiliza as funções básicas para o processamento, como Map, Reduce, Filter, Collect, entre outras;
- **GraphX:** realiza o processamento sobre grafos;
- **SparkSQL:** para a utilização de SQL em consultas e processamento sobre dados;
- **Mlib:** disponibiliza a biblioteca de aprendizado de máquina.

O Spark não conta com um sistema próprio de gerenciamento de arquivos, portanto, precisa ser integrado a um, como o HDFS do Hadoop, como foi sugerido. Mas também é possível utilizá-lo com uma base de dados em *cloud computing*. A arquitetura Spark é definida no Spark Core e é composta principalmente de três componentes principais:

- **Driver Program:** aplicação principal que gerencia a criação e executa o processamento definido pelo programador;
- **Cluster Manager:** administra o *cluster* de máquinas quando a execução for distribuída;
- **Workers:** executam as tarefas enviadas pelo Driver Program.

Os conceitos mais importantes utilizados na programação e no desenvolvimento de soluções com Spark incluem:



- **Resilient Distributed Dataset (RDD)**: funciona como uma abstração do conjunto de objetos distribuídos pelo *cluster*. É o objeto principal do modelo de programação no Spark;
- **Operações**: são as transformações e ações realizadas num RDD;
- **Spark Context**: objeto que representa a conexão da aplicação com o *cluster*. Pode ser utilizado para criar RDDs, acumuladores e variáveis no *cluster*.

TEMA 5 – ANÁLISE E VISUALIZAÇÃO

Apenas recentemente a capacidade de armazenamento e processamento se tornaram suficientes para permitir que dados antes ignorados fossem analisados. Entretanto, além dos componentes tecnológicos, o analista de dados deve ser capaz de identificar quais dados se deve utilizar, como integrá-los, quais perguntas serão úteis para a tomada de decisão, e qual a melhor maneira de apresentar os resultados obtidos da análise.

5.1 Análise de dados

A extração de informações úteis pela análise de dados não é uma tarefa simples. Em muitos casos, os dados podem ter informações incompletas, inconsistências, caracteres indesejados, estarem corrompidos, duplicados, em formato inadequado, e outros tipos de problema. Segundo Marquesone (2016), cerca de 80% do tempo da análise de dados é utilizado apenas para limpar e preparar os dados.

Durante a análise, podemos constatar a importância da qualidade dos dados utilizados pois, sem um processo de inspeção, muitos dados incorretos podem ser descartados. No entanto, mesmo que a qualidade seja garantida, a busca por padrões nos dados ainda é um grande desafio. É muito fácil analisá-los de forma errada, ao não se identificar corretamente relações de correlação e causalidade, e propagar erros que invalidem toda a análise. Por fim, é necessário validar todos os resultados gerados pela análise dos dados antes de serem utilizados.



5.2 O processo de análise de dados

A análise de dados inclui como atividades a identificação de padrões nos dados, sua modelagem e classificação, detecção de grupos, entre muitas outras. Para isso, utilizamos técnicas matemáticas, estatísticas e de aprendizado de máquina. O aprendizado de máquina pode ser muito útil na automatização da construção de modelos analíticos. Pode-se extrair informações úteis e padrões ocultos em conjuntos massivos de dados.

Os processos de análise de dados podem ser definidos de diversas formas. Uma delas seria por um padrão aberto conhecido pela sigla CRISP-DM (*cross-industry standard process for data mining*), que define as seguintes fases e tarefas:

- **Entendimento de negócio:** determinar os objetivos de negócio, seu contexto e critérios de sucesso; avaliar recursos disponíveis, riscos e contingências, definir terminologias e calcular custos e benefícios; determinar os objetivos da mineração de dados e seus critérios de sucesso; e produzir um plano de projeto. Nessa fase são definidas as perguntas, os objetivos e os planos;
- **Compreensão dos dados:** fazer a coleta inicial e descrever os dados; fazer análise exploratória; e verificar a qualidade dos dados. O objetivo é entender a estrutura, atributos e contexto em que os dados estão inseridos;
- **Preparação dos dados:** descrever o conjunto, selecionar, filtrar e limpar os dados, e minimizar a geração de resultados incorretos; construir dados (atributos derivados, registros gerados); integrá-los (mesclagem ou redução); formatá-los e estruturá-los;
- **Modelagem dos dados:** selecionar técnicas de modelagem; projetar testes; definir e construir o modelo de dados, seus parâmetros e sua descrição; e validar o modelo e definir os parâmetros a revisar. Para construir o modelo, utilizamos tarefas de algoritmos de extração de padrões que podem ser agrupadas como atividades descritivas ou preditivas;



- **Avaliação do modelo:** avaliar os resultados do modelo; revisar processos; e determinar os passos seguintes. Avalia-se a precisão dos resultados gerados com os modelos de dados;
- **Utilização do modelo:** planejar a entrega; planejar o monitoramento e a manutenção; produzir relatório final; e documentar e revisar o projeto. Os modelos aprovados são então utilizados e monitorados.

5.3 Visualização de dados

Obtidos os resultados pela análise dos dados, ou até antes disso, é interessante poder comunicar as informações obtidas. Ao mesmo tempo, é importante entender quais informações são mais relevantes e qual a melhor forma de apresentá-las com clareza. Uma vez que nós, como humanos, somos dotados de grande percepção visual, a representação dos dados de forma gráfica é muito eficiente para expressar as informações que obtivemos dos dados. Assim, a visualização de dados é definida como **a comunicação da informação utilizando representações gráficas**.

5.3.1 Visualização exploratória

A análise de dados requer que eles sejam avaliados de forma detalhada. Para melhorar a compreensão deles nessa etapa, é possível usar a visualização exploratória, que auxilia na identificação de estruturas das variáveis, das tendências e de relações, permitindo até mesmo a detecção de anomalias nos dados.

Existem muitas formas de representar dados graficamente. Cada uma delas é capaz de exibir certo nível de detalhamento ou destacar características específicas. Inclusive é muito comum que o analista de dados use diferentes tipos de gráfico para estruturar os dados conforme necessário, para melhorar sua compreensão.

5.3.2 Visualização explanatória

Quando o analista já tiver resultados concretos, ele está pronto para comunicar suas percepções. Essa etapa é definida como *visualização explanatória*. Durante ela, o interesse do analista é destacar os detalhes



importantes, de forma a comunicar os resultados obtidos de um grande volume de dados em informações mais concisas e de fácil compreensão no formato de uma interface visual. Em muitos casos, essa informação permite revelar tendências e desvios que podem servir de apoio a tomadas de decisão.

Uma interface visual permite que o leitor veja características específicas e detalhadas dos dados. Para isso, existem muitos atributos que devem ser analisados durante a criação dos gráficos. Cada tipo de gráfico pode ser mais adequado para comunicar certa informação a respeito de seus dados. Por exemplo, gráficos de colunas, barras, áreas circulares, linhas e de dispersão são mais adequados para comparar valores, enquanto gráficos de dispersão, histogramas e gráficos de área são mais adequados para destacar a distribuição de um conjunto de dados. Para cada necessidade, existe algum tipo de gráfico apropriado.

5.4 A visualização de dados

A literatura mostra que existem sete etapas para a visualização de dados, incluindo algumas que fazem parte das etapas de coleta, armazenamento, processamento e análise de dados.

1. **Aquisição:** etapa em que ocorre a coleta de dados;
2. **Estruturação:** define-se a estrutura em que os dados são padronizados;
3. **Filtragem:** dados incorretos, incompletos ou desinteressantes para a análise são removidos;
4. **Mineração:** parte da etapa de análise de dados que extrai informações dos dados;
5. **Representação:** etapa da análise exploratória que gera um modelo visual básico de dados;
6. **Refinamento:** técnicas gráficas para tornar a visualização mais eficiente;
7. **Interação:** inclui funcionalidades que oferecem melhor experiência para o leitor.

Existem ferramentas que auxiliam a visualização de dados a ponto de automatizar muitas etapas. Como vimos, ela pode ser importante durante a análise dos dados, pois contribui para resultados com maior precisão, ou ainda atende soluções que necessitam de visualização em tempo real.



FINALIZANDO

Neste capítulo, vimos uma introdução aos principais fundamentos que compõem o conceito de Big Data. Iniciamos com uma breve contextualização histórica da evolução da geração de dados e dos avanços tecnológicos que aumentaram o volume de dados a uma escala imensa, permitindo que dados anteriormente ignorados ou descartados passassem a ser analisados de forma cada vez mais detalhada.

Vimos que uma solução Big Data pode ser definida pela coleta, processamento, análise e visualização de um volume muito grande de dados confiáveis (quanto à veracidade) e variados (tanto no formato quanto na origem), com a velocidade necessária para gerar o valor adequado à solução. Vimos um pouco sobre os processos de coleta, armazenamento, processamento, análise e visualização de dados, passando por noções básicas de bancos de dados não relacionais (NoSQL), Hadoop, Spark, entre outros.



REFERÊNCIAS

TAURION, C. **Big Data**. Rio de Janeiro: Brasport, 2013.

MARQUESONE, R. **Big Data**: técnicas e tecnologias para extração de valor dos dados. São Paulo: Casa do Código, 2016.



BIG DATA

AULA 3

Prof. Luis Henrique Alves Lourenço



CONVERSA INICIAL

Nesta aula, aprofundaremos a discussão sobre os componentes Hadoop responsáveis pela integração da ferramenta de Big Data com Bancos de Dados Relacionais, Bancos de Dados Não Relacionais, também conhecidos como *Bancos NoSQL*. Além disso, vamos conhecer as ferramentas capazes de realizar consultas aos dados independentemente de onde eles se situam, inclusive misturando os de armazenamento de forma transparente ao usuário.

TEMA 1 – HIVE

A primeira característica que podemos destacar a respeito do Hive é que ele se aproveita de uma sintaxe semelhante ao SQL. O Hive foi projetado para ser uma aplicação de *Data Warehouse open source* que opera sobre componentes do Hadoop, como HDFS e MapReduce. *Data Warehouse* é uma categoria de aplicações responsáveis por armazenar dados de diversos sistemas em um repositório único. Os dados de tais sistemas são transformados para serem formatados de acordo com um padrão específico. Dessa forma, o Hive é capaz de traduzir consultas em um dialeto de SQL, o Hive SQL, para tarefas do MapReduce ou Tez. Por sua semelhança com consultas SQL, o HiveQL é uma ferramenta muito poderosa e fácil de utilizar.

O Hive permite também a requisição de consultas de maneira interativa por meio de um prompt de comando ou um terminal, assim como qualquer banco de dados relacional. Uma vez que o Hive opera sobre uma estrutura de Big Data, ele é altamente escalável e adequado para trabalhar com grandes volumes de dados.

Por essas características, o Hive é um componente muito adequado para ferramentas OLAD (*OnLine Analytical Processing*), utilizadas para analisar dados multidimensionais interativamente por diversas perspectivas, o que permite consultas analíticas complexas e especializadas com rápido tempo de execução. Sem o Hive, isso é possível apenas por meio de implementações de funções MapReduce em Java, sendo muito mais complexo do que as consultas em HiveQL.

É válido destacar que os dados utilizados em consultas HiveQL não estão em um banco de dados relacionais com tabelas bem definidas. Portanto,



algumas das funcionalidades comuns de bancos de dados não existem da mesma maneira. Por exemplo, as consultas não realizam transações e, por isso, operações como *update*, *insert* e *delete* não funcionam da mesma forma. Portanto, podemos dizer que a grande desvantagem do uso de Hive é a sua limitação quanto às operações possíveis. Componentes como Pig e Spark oferecem uma quantidade maior de recursos e permitem realizar operações mais complexas.

Ainda assim, HiveQL implementa grande parte das operações SQL e algumas extensões. Uma das possibilidades permitidas pelo HiveQL é o armazenamento do resultado de uma consulta em uma estrutura chamada de *view*, que possibilita que consultas futuras acessem tais dados como se estivessem em tabelas de um banco. Além disso, o Hive permite as quatro operações básicas de transações em bancos de dados definidas pelas propriedades ACID: Atomicidade, Consistência, Isolamento e Durabilidade.

1.1 Arquitetura Hive

O Hive oferece a possibilidade de implementar extensões por meio de funções definidas pelo usuário, pelo servidor Thrift (para C++, Python, Ruby, e outras linguagens) e por drivers JDBC e ODBC. Além disso, é possível efetuar consultas por uma interface de linha de comando (CLI), ou pela *Hive Web Interface (HWI)*.

Todos os comandos e consultas são recebidos pelo *Driver*, que compila a entrada, otimiza a computação necessária e executa os passos, originalmente com tarefas MapReduce. O Hive não gera funções em Java para o MapReduce. Em vez disso, ele utiliza Mappers e Reducers genéricos, sequenciados por planos de trabalho escritos em XML. Dessa forma, o Hive se comunica com a aplicação mestre para iniciar a execução pelo MapReduce ou Tez.

Bancos de Dados Relacionais originalmente foram projetados baseados em uma técnica para o armazenamento de dados conhecida por *schema on write*. Essa estratégia define que as partes dos dados precisam se ajustar a um padrão ou um plano no momento da escrita. Com o tempo, uma característica dessa estratégia se tornou evidente: ela é muito restritiva. Perde-se muito tempo ajustando os dados à estrutura definida. Contudo, a quantidade de dados semiestruturados e não estruturados que precisam ser analisados é cada vez maior, o que causa um aumento dessa desvantagem.



Sendo assim, o Hive inverte a lógica e faz uso do conceito contrário: *scheme on read*. A estrutura dos dados, também conhecida como *esquema*, só é definida durante a leitura dos dados. Isso permite que os dados sejam armazenados do modo como são recebidos, estruturados ou não. Assim, os dados são estruturados apenas durante o processo de leitura de maneira muito mais específica para a forma como ele vai ser utilizado. Por essa característica, podemos dizer que essa estratégia é adequada para a utilização dos dados contidos no HDFS, uma vez que o sistema de arquivos distribuído não exige que os dados armazenados sejam estruturados. Por isso, o Hive consegue simular a utilização de um banco de dados que se aproveita das características de um sistema de arquivos distribuídos, como é o HDFS.

Os dados armazenados pelo Hive podem ser particionados em subdiretórios. Isso permite uma grande otimização, uma vez que limita a quantidade de dados em que a consulta ocorre. O Hive mantém em um banco de dados relacional (Derby ou MySQL) os metadados necessários para suportar os esquemas que definem como os dados devem ser lidos e os particionamentos que definem como eles devem ser organizados no sistema.

1.2 HiveQL

HiveQL é uma linguagem para consultas implementada pelo Hive. Como todos os dialetos SQL, não é totalmente compatível com uma revisão particular do padrão ANSI SQL. É muito parecido com MySQL, levando algumas pequenas diferenças. HiveQL não implementa os comandos *insert*, *update* e *delete* em linha, além de não implementar transações.

Hive implementa algumas extensões como as que oferecem melhorias de desempenho no contexto do Hadoop e as que servem para integrar extensões customizadas ou programas externos.

1.2.1 Database

O conceito de *database* em Hive consiste em um catálogo (*namespaces*) de tabelas. Essa definição é muito útil para evitar nomes de tabelas duplicadas. Se nenhuma database for especificada, uma database *default* é utilizada por padrão. A maneira mais simples de criar uma database é da seguinte forma:

```
hive> CREATE DATABASE nova_database;
```



Assim, se *nova_database* já existir, o Hive retornará um erro. Para evitá-lo, é possível utilizar a cláusula IF NOT EXISTS para não criar uma database que já existe. Isso é muito interessante para ser utilizado em *scripts* que precisam criar databases durante a execução:

```
hive> CREATE DATABASE IF NOT EXISTS nova_database;
```

O uso da palavra reservada SCHEMA pode substituir DATABASE em todos os comandos relacionados à database. A qualquer momento se pode verificar as databases existentes com a operação SHOW:

```
hive> SHOW DATABASES;
default
nova_tabela
```

Pode-se usar expressões regulares para filtrar as databases desejadas, utilizando as palavras reservadas LIKE ou RLIKE da seguinte forma:

```
hive> SHOW DATABASES LIKE 'n.*';
nova database
```

Para cada database, é criado um diretório. As tabelas de cada database são armazenadas em subdiretórios do diretório da database. A exceção, porém, são as tabelas da database default que não têm um diretório próprio. O diretório da database é criado no diretório especificado pela propriedade *hive.metastore.warehouse.dir*. Por padrão, o diretório especificado é o /user/hive/warehouse. Ao criar a database *nova_database*, o Hive irá criar o diretório /user/hive/warehouse/nova_database.db. É possível forçar a localização do novo diretório utilizando a palavra reservada LOCATION.

Para remover a database, utiliza-se a operação DROP:

```
hive> DROP DATABASE IF EXISTS nova_database;
```

O IF EXIST é opcional e, assim como no CREATE serve para evitar erros, caso a database não exista. Por padrão, o Hive não permite que uma database com contenha tabelas seja removida. Dessa forma, é necessário remover as tabelas primeiro, ou utilizar a palavra reservada CASCADE, que faz isso automaticamente.



1.2.2 Criação de Tabelas

Em Hive, a declaração CREATE TABLE segue os padrões de SQL estabelecidos. Mas, além disso, o Hive estende o suporte à flexibilidade em relação aos lugares em que os dados podem ser armazenados, os formatos de dados, entre outras extensões. É possível definir uma database prefixando seu nome antes do nome da tabela, separados por um ponto:

```
hive> CREATE TABLE IF NOT EXISTS mydb.funcionarios;
```

Da mesma forma, podemos utilizar a expressão IF NOT EXISTS para suprimir os erros no caso de já existir uma tabela com esse nome. No entanto, dessa forma, o Hive não o avisará se o esquema da tabela existente for diferente da tabela que está se tentando criar. Caso você deseje criar um esquema para essa tabela, você pode removê-la com DROP e recriá-la. No entanto, todos os dados serão perdidos. Dependendo da situação, pode ser possível executar algumas operações de ALTER TABLE para modificar o esquema de uma tabela existente. Também é possível utilizar a expressão LIKE para copiar o esquema de uma tabela previamente existente.

Para popular tabelas com dados, podemos utilizar a operação LOAD DATA e mover os dados de um sistema distribuído para o Hive. Na prática, o comando apenas indica que os dados serão utilizados pelo Hive, e os dados serão movidos para onde precisam estar. E podemos utilizar a operação LOAD DATA LOCAL para copiar os dados do sistema de arquivos local para o Hive. Dessa forma, as operações efetuadas nas tabelas não se refletem nos dados originais. É possível obter o mesmo efeito utilizando a expressão CREATE EXTERNAL para criar tabelas sem vínculo com os dados de onde elas foram importadas.

Para partitionar os dados de uma tabela, utilizamos a expressão PARTITIONED BY. Dessa forma, os dados armazenados são divididos por uma propriedade específica em subdiretórios:

```
hive> CREATE TABLE funcionarios(
        nome STRING
        setor STRING
    ) PARTITIONED BY (setor STRING)
```



TEMA 2 – INTEGRANDO HADOOP COM BANCO DE DADOS RELACIONAIS

Hive é uma ferramenta poderosa para visualizar dados inseridos no sistema de arquivos distribuídos do Hadoop como se eles estivessem em um banco de dados relacional. No entanto, muitas vezes, os dados que desejamos incluir em nossa análise já estão armazenados em um banco de dados externos ao Hadoop, como MySQL, Postgress e outros. Bancos de dados relacionais, como o MySQL, geralmente têm uma estrutura de armazenamento monolítica, ou seja, os dados estão instalados localmente em apenas um servidor, ao contrário do armazenamento feito pelo HDFS no Hadoop. Por essa característica, são ferramentas mais adequadas à OLTP (*OnLine Transaction Processing*), sistemas que registram todas as transações de determinada operação. São muito utilizados em sistemas financeiros e bancários, pois as transações devem ser atômicas e consistentes. Além disso, tais sistemas, na maioria das vezes, necessitam que o processamento das consultas seja muito rápido, de forma a manter a integridade dos dados em um ambiente com acessos variados e concorrentes. No entanto, devido a sua natureza distribuída, o armazenamento baseado no HDFS tem consultas mais complexas e lentas.

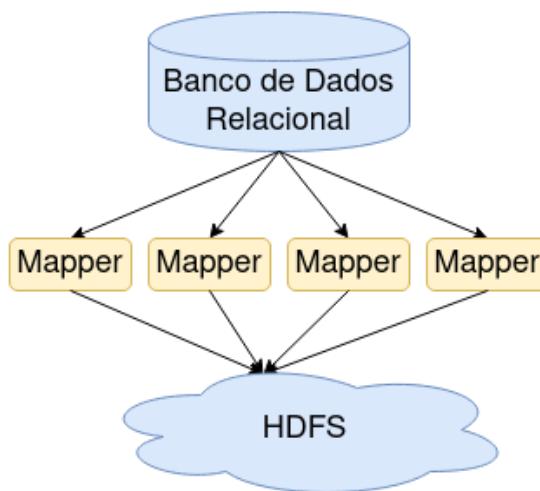
Dessa forma, existem muitos cenários em que dados que operam em bancos de dados relacionais possam ser analisados por si só, ou em conjunto com outros dados. Tornando evidente a necessidade de ferramentas que transferem dados entre fontes de dados estruturados, semiestruturados e não estruturados.

2.1 Importando dados com Sqoop

Sqoop é uma ferramenta projetada para transferir grandes volumes de dados estruturados, semiestruturados ou não estruturados de maneira eficiente entre o Hadoop e aplicações externas. Para isso, ele utiliza o MapReduce ou o Hive para fornecer operações paralelas e tolerantes a falhas. A maioria dos aspectos pode ser customizada. É possível controlar as linhas e colunas específicas a serem importadas; determinar delimitadores, caracteres de escape para as representações de dados baseados em arquivos, assim como o formato dos dados; a quantidade de tarefas MapReduce, entre muitas outras características.



Figura 1 – Fluxo de importação de dados do Sqoop



Fonte: Luis Henrique Alves Lourenço.

Para realizar a importação de dados por meio de um banco de dados MySQL, podemos utilizar o seguinte comando:

```
sqoop import --connect jdbc:mysql://localhost/foobar -m1 --  
driver com.mysql.jdbc.Driver --table foo
```

Assim, podemos utilizar o conector JDBC (*Java DataBase Connector*) como driver que conecta o banco de dados ao Sqoop. O parâmetro `-m` é utilizado para definir a quantidade de Mappers que serão criados para executar a operação. E o parâmetro `--table` é utilizado para especificar a tabela que será importada. O Sqoop também oferece formas de especificar o nome do arquivo criado no HDFS.

Primeiramente, o Sqoop analisa a base de dados e cria os metadados que serão utilizados para recuperar os dados. Em seguida, ele requisita trabalhos de MapReduce para transferir os dados. Cada linha do arquivo criado pelo HDFS representa uma entrada da tabela e valores separados por vírgula para cada coluna. É possível configurar qualquer caractere para fazer a separação dos valores. Além disso, é possível utilizar o parâmetro `--hive` para importar os dados do banco diretamente para o Hive.



2.2 Importação incremental

O Sqoop oferece um modo de importação incremental que pode ser utilizado para retornar apenas as linhas mais novas do que as importadas anteriormente. Dessa forma, é possível manter os dados do Hadoop sincronizados com o banco de dados relacional. Dois modos de importação incremental são suportados pelo Sqoop pelo parâmetro `--incremental`: `append` e `lastmodified`. O modo `append` deve ser utilizado quando dados são inseridos continuamente na tabela importada e uma das colunas armazena um valor que é incrementado. Tal coluna deve ser indicada pelo parâmetro `--check-column`. Assim, o Sqoop importa às linhas que a coluna verificada tem um valor maior do que o especificado pelo parâmetro `--last-value`. A estratégia de importação incremental definida pelo modo `lastmodified` foi projetada para ser utilizada quando os dados da tabela possam ser atualizados, e tal atualização escreve o `timestamp` atual na coluna indicada pelo parâmetro `--check-column`. As linhas com `timestamp` mais recente que o `timestamp` indicado pelo parâmetro `--last-modified` são importadas.

Ao final de uma operação de importação incremental, o valor que deve ser especificado como `--last-value` para a importação seguinte é retornado, de maneira que, ao realizar a importação seguinte, pode ser especificado como `--last-value` para garantir que apenas os dados novos ou atualizados sejam importados. Isso é tratado automaticamente ao criar uma importação incremental como um trabalho salvo (*saved job*). O `sqoop-job` é a ferramenta do Sqoop para armazenar os parâmetros de uma operação e poder reutilizá-la. Essa é a forma mais adequada de criar um mecanismo recorrente de importação incremental e, assim, manter os dados do Hadoop e do banco de dados sincronizados.

TEMA 3 – BANCO DE DADOS NÃO RELACIONAIS

Como vimos anteriormente, bancos de dados relacionais ainda hoje são adequados para muitas aplicações. Por isso, são uma fonte muito grande de dados valiosos. Graças a tecnologias como Hive e Sqoop, podemos integrá-los e fazer uso dos dados dessas aplicações para analisar em conjunto com uma grande quantidade de dados.

No entanto, como sabemos, bancos de dados relacionais e consultas SQL apresentam limitações, principalmente quando queremos analisar grandes



volumes de dados distribuídos em *clusters*. É para esse tipo de situação que se torna muito importante entendermos a tecnologia por trás dos bancos de dados não relacionais, que renunciam à linguagem de consulta SQL e são capazes de processar dados semiestruturados e não estruturados. Bancos de dados não relacionais, também conhecidos como NoSQL (*No SQL* ou, em alguns casos, *Not Only SQL*), são capazes de escalar horizontalmente de forma ilimitada.

Em NoSQL, utilizamos o conceito de teorema CAP para descrever o comportamento de um sistema distribuído. Esse teorema ajuda a explicar a relação que cada estratégia de banco de dados NoSQL adota quando eles têm a necessidade de requisitar uma operação de escrita seguida de uma operação de leitura, de forma que não necessariamente tais operações vão utilizar os mesmos nós do cluster. Assim, o teorema prevê que um sistema distribuído pode garantir apenas dois de três comportamentos. São eles: consistência, disponibilidade (*availability*) e tolerância a falhas ou tolerância a partição (*partition tolerance*). A consistência garante que o sistema sempre vai ler o dado mais atualizado, ou seja, tão logo ocorra a escrita de um dado no sistema, este dado pode ser lido por meio de qualquer nó do sistema. A disponibilidade implica que o cliente deve ter acesso aos dados mesmo na ocorrência de falhas. E a tolerância a falhas garante que mesmo que partes do sistema se encontrem desconectados por uma falha na rede, o sistema continua operando normalmente.

3.1 NoSQL

Os volumes de dados em muitos casos podem ser tão grandes que os bancos de dados relacionais não suportam escalar mais, pois dependem majoritariamente da escalabilidade vertical. Como já vimos, a escalabilidade vertical não tem a capacidade de escalar de forma ilimitada, pois depende dos avanços tecnológicos. Mesmo fazendo uso de diversas técnicas, como denormalização dos dados, que aumenta a redundância do banco, mas pode causar riscos à integridade dos dados; o uso de múltiplas camadas de cache; visões materializadas (*materialized views*), que é um objeto com os resultados de uma consulta; a utilização de uma configuração mestre-escravo que permite a replicação de um banco; ou a simples divisão dos dados em diferentes bancos. Todas essas técnicas otimizam a forma com que os bancos de dados relacionais lidam com grandes volumes de dados. Mas, ao mesmo tempo, seu uso pode



deixar o sistema excessivamente complexo. Podemos dizer, então, que para um volume muito grande de dados, a escalabilidade vertical dos bancos de dados relacionais pode não ser o suficiente.

Uma das principais características dos bancos de dados não relacionais, ou NoSQL, é a de não utilizar o modelo relacional. O modelo relacional se baseia no princípio de que os dados estão em tabelas, na lógica de predicados e na teoria de conjuntos. Os bancos de dados não relacionais, por outro lado, permitem que os dados sejam acessados de forma não relacionais, ou seja, sem precisar obedecer às regras da lógica de predicados e da teoria de conjuntos. Dessa forma, os modelos de bancos de dados não relacionais não são capazes de garantir transações com propriedades ACID. No entanto, não há um modelo único. Os diversos modelos de bancos de dados não relacionais são classificados em:

- **Bancos de dados de esquema chave-valor** – são bancos que utilizam um método de chave-valor para armazenar os dados também conhecido como *array associativo*. Os dados são armazenados em um conjunto de pares chave-valor em que a chave é um identificador exclusivo que permite a busca pelos valores. Em muitas implementações, a chave e os valores podem ser qualquer tipo de dado. Tanto a chave quanto os valores podem ser texto, números, estruturas de dados complexas ou até mesmo objetos binários. No entanto, chaves muito grandes podem ser ineficientes. Além disso, as chaves têm um modelo discretamente ordenado que as mantêm em ordenação lexicográfica. Essa característica é muito útil quando os dados estão distribuídos pelo *cluster*. É o modelo NoSQL mais simples. Ele permite rápido acesso aos dados pela chave por meio de uma API simplificada.
- **Bancos de dados baseados em documentos** – são bancos que utilizam o conceito de documentos para armazenar os dados. Cada implementação pode divergir a respeito de como os documentos são definidos, porém, em geral, todos assumem que documentos devem ser capazes de encapsular e codificar os dados. Para isso, formatos como XML, YAML, JSON e BSON são comuns. Há um esquema altamente flexível e com solução muito adequada para dados semiestruturados. Cada documento é endereçado por uma chave única, de maneira muito parecida com os bancos de esquema chave-valor. Além disso, ele oferece



uma API capaz de consultar documentos baseado em seus conteúdos, e tais bancos normalmente utilizam uma terminologia diferente. Comparado com os bancos de dados relacionais, podemos dizer que coleções são análogas a tabelas, documentos são registros (ou linhas), e as colunas são campos. Os campos são bastante diferentes neste caso, pois cada documento em uma coleção pode apresentar campos completamente diferentes.

- **Bancos de dados baseado em colunas** – são bancos de dados que utilizam o conceito de famílias de colunas para armazenar os dados em registros de tabelas. Uma família de colunas consiste em um conjunto de colunas opcionais. Dessa forma, cada registro não precisa dispor de dados nas mesmas colunas que os demais. É uma estratégia muito útil para dados esparsos. Cada valor é indexado por linha, coluna e *timestamp*.
- **Bancos de dados baseados em grafos** – são bancos que utilizam os conceitos desenvolvidos na teoria dos grafos. Um grafo é composto de dois elementos: um nó e uma relação. Cada nó representa uma entidade (que é parte do dado, como uma pessoa, um lugar, uma coisa, uma categoria etc.). E cada relação é a representação de como dois nós estão relacionados. Neste modelo de bancos, a relação entre os dados é normalmente mais relevante do que os dados em si. Dessa forma, é importante que as relações entre os dados sejam persistentes durante todo o ciclo de vida dos dados. Alguns bancos de dados baseados em grafos utilizam um tipo de armazenamento especificamente projetado para estruturas de grafos. No entanto, outras tecnologias podem utilizar bancos relacionais, baseados em documentos ou colunas, como camada de armazenamento em uma estrutura lógica de grafos. Além disso, os bancos de dados baseados em grafos implementam um modelo de processamento, chamado de adjacência livre de índices. Com isso, dados relacionados entre si apontam fisicamente um para o outro no banco de dados.

3.2 HBase

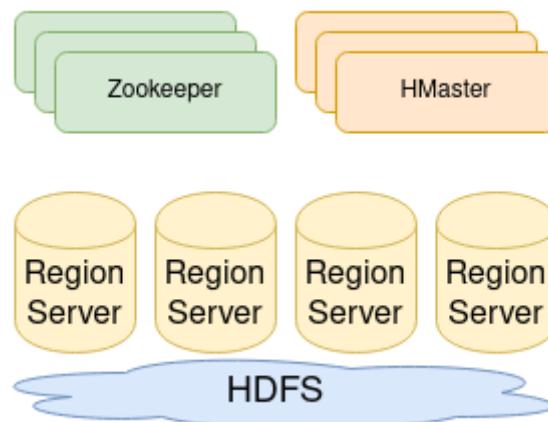
O HBase é um banco de dados não relacional projetado para utilizar o sistema de arquivos distribuído Hadoop (HDFS) da mesma forma que o BigTable



é utilizado sobre o Google File System. BigTable é um sistema de armazenamento distribuído para gerenciar dados estruturados, projetado para arquivos muito grandes (*petabytes* de dados distribuídos em milhares de computadores). Dessa forma, o HBase é o seu equivalente *open source* projetado para o Hadoop. O HBase não utiliza uma linguagem de consulta como o SQL. No entanto, utiliza uma API Java baseada no padrão CRUD. Sua arquitetura permite garantir acessos de escrita e leitura fortemente consistentes, em vez de eventualmente consistentes, ou seja, um dado inserido pode ser lido de forma correta instantaneamente de qualquer parte do *cluster*. Além disso, o HBase suporta processamento massivamente paralelizado por meio do MapReduce.

Tabelas são distribuídas pelo *cluster* por um conceito conhecido por *Regiões (regions)*. Regiões são o elemento básico da disponibilidade e distribuição das tabelas, e são gerenciados pelos *RegionServers*, executados nos *DataNodes*. É o componente responsável pelo particionamento automático dos dados de cada tabela pelo *cluster* que podem ser divididos e redistribuídos à medida que os dados crescem.

Figura 2 – Arquitetura do HBase



Fonte: Luis Henrique Alves Lourenço.

3.2.1 Arquitetura do HBase

O cliente HBase consulta a lista de todas as regiões no sistema, que é armazenada pelo Zookeeper, para localizar o RegionServer que oferece a parcela de registros em que se encontra o dado de interesse. Após localizar o RegionServer, o cliente se comunica com ele para requisitar as leituras e escritas



desejadas. Internamente, o cliente HBase utiliza um registro de conexões para obter o endereço do servidor mestre ativo, localização das regiões, e a identificação do *cluster*.

HMaster é a implementação do servidor mestre. O servidor mestre é responsável por monitorar todos os RegionServers do *cluster* e opera como interface para todas as alterações de metadados, como esquemas e particionamentos. Em um *cluster*, ele normalmente executa em um *NameNode*. O HBase pode ser configurado para executar mais de um servidor mestre. Dessa forma, um deles deve ser eleito como ativo. Esse gerenciamento ocorre por meio do Zookeeper, como já vimos.

3.2.2 Modelo de dados

Uma linha (*row*), ou registro, no HBase, é composta por uma chave e uma ou mais colunas com valores associadas a elas. As linhas são ordenadas alfabeticamente por suas chaves à medida que são armazenadas. Por esse motivo, a modelagem da chave de cada linha é muito importante, pois é muito relevante para manter dados relacionados em regiões próximas.

Colunas são agrupadas em conjuntos de famílias de colunas. Uma coluna consiste em uma família de colunas e um qualificador, normalmente utilizado signo dois pontos (:). Cada linha tem as mesmas famílias de colunas, mas não precisa que as mesmas colunas sejam preenchidas para todas as linhas. O qualificador é o que determina cada uma das colunas em uma família de colunas. Por exemplo, se temos uma família de colunas chamada *conteúdo*, podemos ter como qualificadores de coluna *conteúdo:html* ou, ainda, *conteúdo:pdf*.

Célula é a combinação de linha, coluna e *timestamp*. Permite que cada valor armazenado em HBase tenha versões.

Esses conceitos são expostos pela API do HBase ao cliente. A API para manipulação de dados consiste em três operações principais: *Get*, *Put* e *Scan*. *Get* e *Put* são específicos para cada linha e necessitam de uma chave. *Scan* é executado sobre um conjunto de linhas. Tal conjunto pode ser determinado por uma chave de início e uma chave de parada, ou pode ser a tabela toda sem chave inicial ou chave de parada. Muitas vezes, é mais fácil compreender o modelo de dados do HBase como um *map* multidimensional.



TEMA 4 – BANCO DE DADOS NOSQL EXTERNOS

Como vimos, os bancos de dados NoSQL têm diversas aplicações. Muitos deles são construídos por meio de estruturas que não incluem o uso de HDFS. Mesmo assim, em muitos casos, pode ser interessante incluir os dados de tais sistemas para serem analisados por meio das ferramentas disponibilizadas pelo ecossistema Hadoop. Neste tema, conheceremos alguns bancos NoSQL que não são projetados utilizando o HDFS, mas que são muito interessantes de serem integrados em sistemas Hadoop.

4.1 Cassandra

Cassandra é um banco de dados NoSQL distribuído e *open source* baseado no modelo de armazenamento chave-valor. Apesar de ser um banco de dados com características NoSQL, ele se diferencia dos demais ao implementar uma linguagem de consultas semelhante ao SQL conhecida por CQL. Os dados são armazenados em tabelas, linhas e colunas. Foi projetado para priorizar confiabilidade, escalabilidade e alta disponibilidade. No que se refere ao teorema CAP, Cassandra junta aos bancos que priorizam a disponibilidade e a tolerância falhas, renunciando à consistência, ou seja, podemos dizer que é um banco de consistência eventual. Cassandra oferece ainda capacidade de replicar dados por todos os nós do *cluster* (importante lembrar que Cassandra não utiliza um *cluster* Hadoop), garantindo a durabilidade e confiabilidade dos dados.

Cassandra suporta transações leves que suportam parcialmente as propriedades de transação ACID. Outra diferença de Cassandra para os bancos relacionais é a falta de suporte a operações JOIN, chaves estrangeiras, e não tem o conceito de integridade referencial, uma vez que se trata efetivamente de um banco de dados NoSQL.

A definição de dados em Cassandra determina que o objeto de nível mais alto é o *keyspace*, que contém tabelas e outros objetos como visões materializadas (*materialized views*), tipos definidos por usuário, funções e agregados. A replicação de dados é gerenciada em nível de *keyspace*.

O modelo de dados implementado em Cassandra foi projetado para ser um modelo de domínio simples e fácil de entender do ponto de vista de bancos relacionais, mapeando tais modelos para um modelo distribuído de tabelas de



dispersão (*hashtable*). No entanto, existem grandes diferenças entre Cassandra e bancos de dados relacionais. Cassandra não tem suporte à operação JOIN, de forma que se for necessário realizar uma operação de JOIN, você terá que fazer isso em um cliente externo, ou denormalizar os dados em uma segunda tabela. Não existe o conceito de integridade referencial em Cassandra. Ao contrário dos bancos de dados relacionais, que precisam trabalhar com dados normalizados, Cassandra opera de forma mais eficiente com dados denormalizados, ou seja, livres de esquemas. Por isso, quando se necessita realizar operações de JOIN, o recomendado é que os dados sejam denormalizados em uma segunda tabela. Essa abordagem facilita a distribuição dos dados pelo *cluster*. Diferentemente dos bancos relacionais que utilizam a estratégia de *schema on write*, Cassandra, assim como outros bancos NoSQL, utiliza a estratégia de *schema on read*, pois, como vimos, é muito mais eficiente armazenar dados denormalizados em bancos distribuídos.

4.1.1 Protocolo Gossip

Cassandra utiliza uma arquitetura de *cluster* descentralizado, ou seja, que não depende de um nó mestre para gerenciar os demais, evitando, dessa forma, que exista um único ponto crítico que possa afetar a disponibilidade do sistema. Cassandra implementa o protocolo Gossip para a comunicação entre os nós, descoberta de pares e propagação de metadados. Tal protocolo é responsável pela tolerância a falhas, eficiência e disseminação confiável de dados. Cada nó pode disseminar metadados para os demais nós, tais como informações de pertencimento ao *cluster*, *heartbeat*, e ao estado do nó. Cada nó mantém informações sobre todos os nós.

O protocolo Gossip executa uma vez por segundo para cada nó trocando informações de até três outros nós no cluster. Uma vez que o sistema é descentralizado, não há nó coordenando os demais. Cada nó seleciona de forma independente de um a três nós para trocar informações. E é possível que algum nó selecionado esteja indisponível. A troca de mensagens no Gossip funciona de maneira muito parecida com um *three-way-handshake* do TCP. Importante destacar que o protocolo gera apenas uma quantidade linear de tráfego de rede, uma vez que cada nó se comunica com até três outros nós.



4.1.2 Integração com Hadoop

Cassandra tem uma integração nativa com componentes do Hadoop. Inicialmente, apenas o MapReduce apresentava suporte a buscar dados contidos em Cassandra. No entanto, a integração amadureceu significativamente e hoje suporta nativamente o Pig e o Hive. Inclusive, o Oozie pode ser configurado para realizar operações de dados em Cassandra. O Cassandra implementa a mesma interface que o HDFS para fornecer a entrada de dados localmente.

4.2 MongoDB

O MongoDB é um banco de dados NoSQL com suporte corporativo, mas que se mantém *open source* e bem documentado, e modelo de dados baseado em documentos. É um banco que, em relação ao teorema CAP, privilegia a consistência dos dados e a tolerância a falhas. E, portanto, é um banco que pode estar eventualmente indisponível. Seu modelo de dados baseado em documentos tem uma estrutura muito semelhante ao formato JSON. Em MongoDB, dizemos que coleções são equivalentes às tabelas dos bancos relacionais e documentos são como os registros (ou linhas). Uma de suas características é dispor de um esquema muito flexível. Diferentemente dos bancos relacionais, o MongoDB não exige que se defina um esquema antes de inserir dados em uma tabela. Assim, cada documento em uma coleção pode ter campos completamente diferentes entre si. E os tipos de dados podem ser diferentes entre documentos de uma coleção. Além disso, é possível alterar a estrutura de cada documento acrescentando campos novos, removendo campos existentes, ou modificando os tipos de dados contidos em um campo. No entanto, na prática, os documentos de uma coleção compartilham uma estrutura similar. E o MongoDB permite a definição de regras para validar esquemas.

Uma vez que temos um modelo de dados muito flexível, precisamos ficar atentos à modelagem dos dados. Dessa forma, é muito importante definir como os dados serão estruturados e como serão representados os relacionamentos entre os dados. Para isso, temos modelos de dados embutidos ou referenciados.

Os modelos de dados embutidos, também conhecidos como *denormalizados*, armazenam pedaços de informação relacionada em um mesmo documento. Em geral, se utiliza quando as entidades apresentam uma relação



de pertencimento, ou uma relação de um-para-muitos entre si. Como resultado, é necessário menos consultas para realizar operações comuns. Dessa forma, obtém-se melhor desempenho de operações de leitura do banco, assim como a possibilidade de se obter dados relacionados com apenas uma operação. Ainda, permite que sejam atualizados dados relacionados com apenas uma operação de escrita. Em muitos casos, o uso de modelos de dados embutidos tem o melhor desempenho.

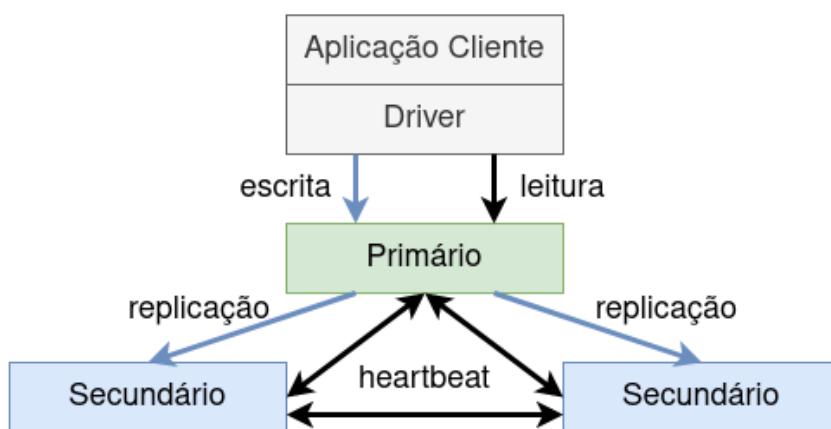
A alternativa aos modelos de dados embutidos são os modelos de dados normalizados. Tais modelos descrevem o relacionamento entre entidades utilizando referências entre documentos. Em geral, são utilizados quando embutir os dados gera uma duplicação de dados ineficiente, em que a eficiência de leitura não é suficiente para superar as implicações causadas pela duplicação; para representar relacionamentos muitos-para-muitos complexos; ou para modelar grandes conjuntos de dados hierárquicos. O MongoDB oferece funções de agregação para unir dados em diferentes coleções por meio de referências.

4.2.1 Conjuntos de replicação

O MongoDB tem uma arquitetura baseada em um único nó mestre. Isso significa que um único nó é responsável pela entrada de dados no banco e que, se esse único nó ficar indisponível, o banco inteiro também estará. Para contornar essa situação, o MongoDB implementa um conjunto de nós para replicar o nó mestre e o substituir em caso de indisponibilidade. Em MongoDB, um conjunto de replicação (*replica set*) é um grupo de processos *mongod* que mantém o mesmo conjunto de dados com múltiplas cópias em diferentes servidores. Os conjuntos de replicação oferecem redundância e melhoram a disponibilidade do banco.



Figura 3 – Arquitetura de conjuntos de replicação



Fonte: Adaptado de MongoDB.

Um conjunto de réplicas contém vários nós que armazenam cópias dos dados e, no máximo, um nó árbitro. Apenas um dos nós que armazenam dados é escolhido como o nó primário, que vai receber a conexão da aplicação cliente, enquanto os demais nós permanecem como nós secundários, que apenas replicam os dados do nó primário. Para isso, os nós secundários aplicam as operações armazenadas no *log (oplog)* do nó primário em seu próprio conjunto de dados de maneira assíncrona. Quando o nó primário se torna indisponível por alguma falha, um novo nó primário é eleito entre os nós secundários. Enquanto ele não for eleito, escritas no banco são desabilitadas. Para eleger um novo nó primário, a maioria dos nós do conjunto de replicação deve concordar sobre quem é o novo nó primário. Se o quórum de nós votantes for menor que a metade do total de nós votantes, não é possível eleger um nó primário e o banco permanece em modo somente-leitura. O ideal é manter um número ímpar de nós, pois, em caso de partição de rede, se o conjunto total de nós for ímpar, apenas o subconjunto que tiver mais do que a metade dos nós poderá receber escritas no banco. Para os casos em que não é possível ter um número ímpar de nós, é possível utilizar um nó árbitro, que não armazena réplicas dos dados e não pode ser escolhido como nó primário. O nó árbitro apenas serve para votar na escolha de um novo nó primário. A eleição se dá da seguinte forma: primeiro são selecionados os nós com maior prioridade. Se mais de um nó for selecionado, então, é escolhido o nó que estiver mais atualizado em relação ao nó primário indisponível. Se o nó primário indisponível voltar a estar disponível, uma nova eleição não é realizada, e o nó volta como um nó secundário.



4.2.2 Particionamento (*sharding*)

Os nós secundários não aumentam a escalabilidade do banco. Apenas são utilizados para melhorar a disponibilidade do sistema em caso de indisponibilidade do nó primário. No entanto, é possível combinar múltiplos conjuntos de replicação para aumentar a escalabilidade horizontal do banco de forma distribuída. Um *cluster* particionado (*sharded cluster*) contém os seguintes componentes:

- Partição (*shard*) – cada partição tem um subconjunto dos dados particionados. Cada partição pode ser atribuída a um conjunto de replicação.
- *Mongos* – são instâncias que atuam como uma interface entre a aplicação cliente e o *cluster* particionado para indicar em qual partição encontram-se os dados que se desejam consultar.
- Servidores de configuração (*config servers*) – conjunto de replicação utilizado única e exclusivamente para armazenar metadados e definições de configuração do cluster.

O MongoDB pode ser particionado de maneira que os dados sejam divididos em diversas partições. Os dados podem ser particionados em *chunks* para serem armazenados nas partições, ou podem ser armazenados sem ser particionados em uma única partição. Os dados particionados devem ter uma chave de partição (*sharded key*) para que sejam distribuídos pelas partições. O MongoDB oferece duas estratégias para distribuir os *chunks* de dados pelas partições. Os *chunks* podem ser distribuídos por faixa de valor das chaves de partição, ou por um *hash* calculado com a chave de partição. Ambas as estratégias são utilizadas para balancear a carga de dados pelas partições.

Além disso, o MongoDB implementa um *framework* de agregação que permite executar operações de MapReduce e um sistema de arquivos distribuídos, o GridFS, para acessar os dados do banco. Assim, é possível utilizar o MongoDB como um substituto do Hadoop. O MongoDB também implementa o suporte à integração com Hadoop, Spark, diversas linguagens, além de um conector SQL para realizar consultas SQL, com algumas restrições.



TEMA 5 – MOTORES DE CONSULTAS SQL

Muitos dos bancos de dados NoSQL não implementam o padrão SQL completamente. No entanto, em muitos casos, é necessária a integração entre as ferramentas armazenamento de grandes volumes de dados implementadas pelos bancos não relacionais por aplicações que utilizam consultas SQL. Além disso, a capacidade de unificar consultas para utilizar diversos bancos diferentes pode ser útil em várias aplicações. Para atender a essas demandas, existem diferentes soluções.

5.1 Drill

Drill é um motor de consultas SQL distribuído *open source* e mantido pela Fundação Apache para a exploração de grandes volumes de dados, por meio da combinação de uma variedade de bancos de dados não relacionais e arquivos.

De forma semelhante a bancos como MongoDB e Elasticsearch, o Drill utiliza um modelo de dados em formato JSON que não exige definição de esquema. Ele automaticamente entende a estrutura dos dados. Tal modelo de dados permite a consulta de dados semiestruturados complexos localmente, sem a necessidade de transformar antes ou durante a execução da consulta. Para isso, o Drill fornece extensões ao SQL para realizar consultas em dados aninhados.

Drill suporta a sintaxe do padrão SQL:2003. Isso significa que o Drill suporta diversos tipos de dados, incluindo DATE, INTERVAL, TIMESTAMP e VARCHAR, assim como permite a construção de consultas complexas, como subconsultas correlacionadas e JOIN em cláusulas WHERE. Dessa forma, o Drill é capaz de operar com ferramentas de BI, tais como Tableau, MicroStrategy, QlikView e Excel. É válido lembrar, porém, que operações muito complexas, especialmente envolvendo JOIN, podem não ser muito eficientes.

Uma característica muito importante do Drill é que, além de permitir consultas SQL, isso pode ser feito pelo uso de diversas fontes de dados, como Hive, HBase, MongoDB, sistemas de arquivos (local ou distribuído, como HDFS e S3). Com isso, é possível realizar um JOIN entre uma tabela Hive e uma HBase ou um diretório de arquivos de log. O Drill esconde toda a complexidade de realizar tais operações SQL em diferentes bancos de dados NoSQL e sistemas de arquivos como se fosse um banco SQL.



O Drill também oferece uma API Java para definir funções customizadas, permitindo que o usuário adicione sua lógica de negócios. Além disso, as funções customizadas para Hive também podem ser utilizadas em Drill.

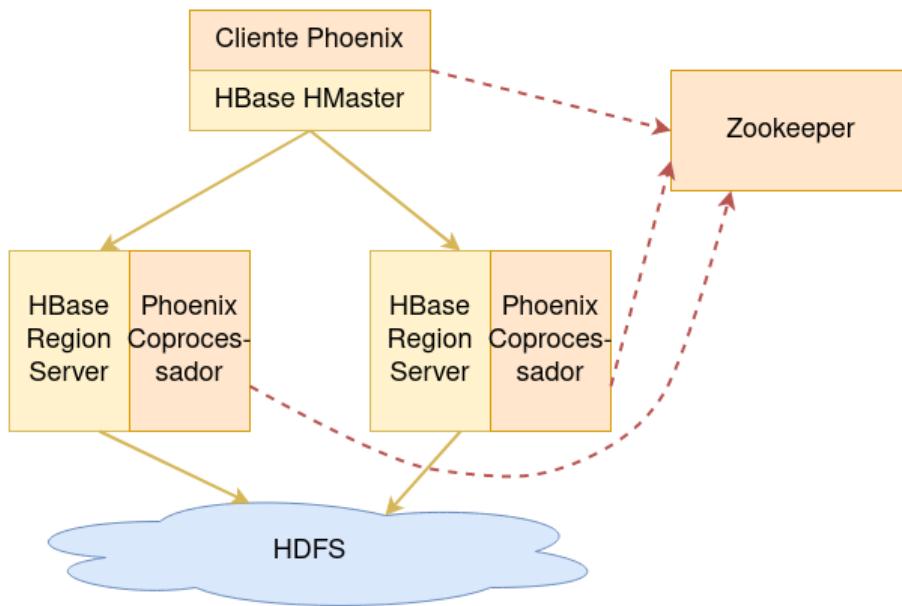
Para processar dados em larga escala, o Drill tem um ambiente de execução distribuída coordenado por um serviço chamado *Drillbit*, responsável por aceitar as requisições dos clientes, processar as consultas e retornar os resultados aos clientes. Ele pode ser instalado em todos os nós de um *cluster* Hadoop para formar um ambiente distribuído em cluster. Dessa forma, o Drill optimiza a execução das consultas por não precisar transferir dados entre os nós. Utiliza o Zookeeper para coordenar os serviços Drillbit em um *cluster*, apesar de poder executar em qualquer ambiente de cluster distribuído, não apenas o Hadoop. No entanto, sua única dependência é o Zookeeper.

5.2 Phoenix

Phoenix é um motor de banco de dados relacional *open source* projetado para operar sobre o HBase, podendo ser utilizado por aplicações que exigem baixa latência e que exigem a capacidade de operar transações OLTP. Isso permite ao Phoenix oferecer transações com propriedades ACID por meio de consultas SQL. O Phoenix traduz consultas SQL para uma série de operações HBase e gerencia a execução dessas operações para produzir um conjunto de resultados que podem ser obtidos por um conector JDBC ou outra interface. Além disso, o Phoenix suporta índices secundários, funções definidas por usuário (inclusive funções HBase), e é possível integrá-lo com MapReduce, Spark, Hive e Flume. Apesar de ser uma camada sobre o HBase, o Phoenix é muito eficiente, uma vez que é capaz de realizar otimizações em consultas complexas automaticamente.



Figura 4 – Arquitetura Phoenix



Fonte: Luis Henrique Alves Lourenço.

Uma aplicação pode se conectar a um cliente Phoenix via JDBC, por interface de linha de comando, por uma API Java. O cliente Phoenix analisa uma consulta SQL e a traduz em operações para o HBase. Como vimos alguns temas atrás, o HBase utiliza-se de servidores de região (*Region Servers*) para distribuir o processamento pelo *cluster*. Para cada *Region Server*, o Phoenix utiliza um componente com o nome de coprocessador para auxiliar na execução e otimização das operações planejadas pelo cliente Phoenix, que serão realizadas pelos *Region Servers*. Além disso, tanto o Phoenix quanto o HBase utilizam o Zookeeper para gerenciar quais *Region Servers* estão disponíveis.

5.3 Presto

Presto é um motor de consultas SQL distribuído *open source* para consultas analíticas interativas capaz de operar em volumes de dados muito grandes. Foi projetado para consultar dados nos locais onde eles estão armazenados, ou seja, interagindo com ferramentas como Hive, Cassandra, MongoDB, Kafka, JMX, PostgreSQL, Redis, Elasticsearch, bancos de dados relacionais, ou mesmo arquivos locais. Ele permite combinar dados de múltiplas fontes. É um motor otimizado para consultas analíticas e data warehouse (OLAP).



Presto não é um banco de dados. De fato, o que ele faz é entender requisições SQL e realizar as operações combinando os dados de diversas fontes de dados. Sua arquitetura é muito similar a um sistema gerenciador de bancos de dados utilizando computação em *cluster*. Pode ser compreendido como um nó coordenador trabalhando em sincronia com múltiplos nós trabalhadores. O Presto analisa operações SQL enviados pelas aplicações clientes e planeja a execução das tarefas dos nós trabalhadores. Os trabalhadores processam conjuntamente os registros das fontes de dados e produzem o resultado que é entregue à aplicação cliente. Para acessar as diversas fontes de dados, Presto implementa conectores específicos para cada ferramenta. Esquemas e referências de dados são armazenados em catálogos (*Catalogs*) e definidos em arquivos de propriedades no diretório de configuração do Presto. Os esquemas em conjunto com os catálogos são a forma utilizada para definir quais tabelas estão disponíveis para serem consultadas.

Presto executa operações SQL e converte tais definições em consultas que serão executadas por um *cluster* distribuído de um coordenador e seus trabalhadores. As operações SQL executadas pelo presto são compatíveis com o padrão ANSI.

FINALIZANDO

Nesta aula, nos aprofundamos nos temas relacionados ao armazenamento em bancos de dados. Começamos entendendo como funciona e para que serve o Hive e sua linguagem semelhante à SQL, HiveQL. Em seguida, aprendemos sobre o Sqoop e como fazer a integração de bancos de dados relacionais com o Hadoop.

Pudemos ver uma explicação sobre os bancos de dados NoSQL, também chamados de *bancos não relacionais*. Em seguida, conhecemos o banco NoSQL HBase, implementado sobre o HDFS, passando por bancos NoSQL que não utilizam a infraestrutura do Hadoop, como o Cassandra e o MongoDB.

Finalizamos conhecendo algumas tecnologias para realizar consultas SQL em bancos NoSQL. Primeiro, vimos como funciona o Drill, projeto da Fundação Apache que implementa um motor de consultas SQL que realiza operações conjuntamente em bancos de dados NoSQL e arquivos em sistemas de arquivos locais ou distribuídos. Conhecemos também o Phoenix, projeto da Fundação Apache que implementa um tradutor de consultas SQL para o HBase.



E, por último, vimos o Presto, um motor de consultas SQL que realiza consultas em diversos tipos de bancos relacionais, NoSQL e sistemas de arquivos.



REFERÊNCIAS

CAPRIOLI, E.; WAMPLER, D.; RUTHERGLEN, J. **Programming Hive.** Data Warehouses and Query Languages for Hadoop. Sebastopol CA: O'Reilly Media, inc., 2012.

JAIN, A. **Instant Apache Sqoop:** Transfer data efficiently between RDBMS and the Hadoop ecosystem using the robust Apache Sqoop. Birmingham UK: Packt Publishing Ltd., 2013.