

Banco de Dados NoSQL

Bootcamp: Analista de Banco de Dados

Ricardo Brito Alves

2021

Banco de Dados NoSQL

Bootcamp: Analista de Banco de Dados

Ricardo Brito Alves

© Copyright do Instituto de Gestão e Tecnologia da Informação.

Todos os direitos reservados.

Sumário

Capítulo 1. Banco de Dados NoSQL	6
Visão geral de NoSQL	6
Propriedades dos bancos de dados.....	6
Escalando um banco de dados relacional	9
Recursos de bancos de dados NoSQL.....	10
Equívocos comuns de bancos de dados NoSQL.....	14
Por que usar o NoSQL?	16
Propriedades dos Bancos de Dados.....	17
Visão geral de NewSQL.....	18
Recursos e técnicas em bancos NoSQL	20
Tipos de bancos NoSQL	24
NoSQL Chave-valor.....	25
NoSQL orientado a documentos.....	26
NoSQL Colunares.....	27
NoSQL Grafos	28
Ferramentas de mecanismo de consulta para NoSQL	29
Motivações no uso do NoSQL	29
Alguns bancos NoSQL.....	31
Capítulo 2. CRUD no MongoDB	33
Mongod.exe	33
Logs	33
Collections	34
Tipos de dados possíveis	35
ObjectId.....	37

CRUD	37
Insert	39
Find	41
Update	41
Delete	42
Operadores	43
Exemplos de outros operadores do MongoDB	43
Agregação.....	45
Capítulo 3. Modelagem e Relacionamentos no MongoDB	48
Normalização de Dados.....	48
Desnormalização de dados.....	49
Principais conceitos Entidade – Relacionamento	50
Linguagem SQL	53
Capítulo 4. Schema e Validation	54
Database e Collections no MongoDB	54
JSON Schema	56
Collection com Validação.....	61
Capped Collection.....	63
ORM, ODM	66
Capítulo 5. Armazenamento de Dados em Nuvem e Sistemas de Arquivos	71
Introdução ao Banco de Dados em Nuvem - DBaaS.....	71
Movimentando seus bancos de dados para a Nuvem	72
Modelos em Nuvem	74
Introdução a AWS.....	76
Google Cloud Platform.....	78

Microsoft Azure	81
Arquitetura Microsserviços.....	82
Sistemas de Arquivos Distribuídos	85
Apache Hadoop	87
MongoDB na Nuvem.....	89
Referências.....	90

Capítulo 1. Banco de Dados NoSQL

Visão geral de NoSQL

Durante a era ponto com, tivemos um uso maciço de bancos de dados relacionais. Em meados dos anos 2000, com a proliferação da Internet, empresas como a Amazon e o Google viram aumentos no tráfego e nos dados. Bancos de dados relacionais, como MySQL, Postgres e Oracle não conseguiam escalar bem. A Amazon criou o SimpleDB e o Google apresentou o BigTable para superar as limitações do RDBMS. A entrada desses dois bancos de dados não relacionais despertou interesse na comunidade de tecnologia.

Em 2009, Johan Oskarsson organizou um encontro para discutir bancos de dados não relacionais distribuídos. Para popularizar este encontro, ele usou uma hashtag #NoSQL no Twitter e isso deu origem aos bancos de dados NoSQL. Neste artigo, vamos fazer um tour pelas limitações do RDBMS, o funcionamento, as capacidades e os equívocos do NoSQL DBS.

Propriedades dos bancos de dados

Propriedades ACID:

- **Atomicidade:** estado em que as modificações no BD devem ser todas ou nenhuma feita. Cada transação é dita como “atômica”. Se uma parte desta transação falhar, toda transação falhará.
- **Consistência:** estado que garante que todos os dados serão escritos no BD.
- **Isolamento:** requer que múltiplas transações que estejam ocorrendo “ao mesmo tempo”, não interfiram nas outras.
- **Durabilidade:** garante que toda transação submetida (commit) pelo BD não será perdida.

Propriedades BASE:

- Basically Available – Basicamente Disponível.
- Soft-State – Estado Leve.
- Eventually Consistent – Eventualmente Consistente.

Uma aplicação funciona basicamente todo o tempo (Basicamente Disponível), não tem de ser consistente todo o tempo (Estado Leve) e o sistema torna-se consistente no momento devido (Eventualmente Consistente).

BDs relacionais trabalham com ACID (Atomicidade, Consistência, Isolabilidade, Durabilidade). BDs NoSQL trabalham com BASE (Basicamente Disponível, Estado Leve, Eventualmente consistente).

CAP:

- Consistency – Consistência.
- Availability – Disponibilidade.
- Partition Tolerance – Tolerância ao Particionamento.

Consistência (Consistent): as escritas são atômicas e todas as requisições de dados subsequentes obtêm o novo valor. Assim, é garantido o retorno do dado mais atualizado armazenado, logo após ter sido escrito. Isso independe de qual nó seja consultado pelo cliente – o dado retornado para a aplicação será igual.

Disponibilidade (Available): o banco de dados sempre retornará um valor desde que ao menos um servidor esteja em execução – mesmo que seja um valor defasado.

Tolerância ao Particionamento (Partition Tolerant) ou Tolerante a Falhas: o sistema ainda irá funcionar mesmo se a comunicação com o servidor for temporariamente perdida. Assim, se houver falha de comunicação com um nó da rede

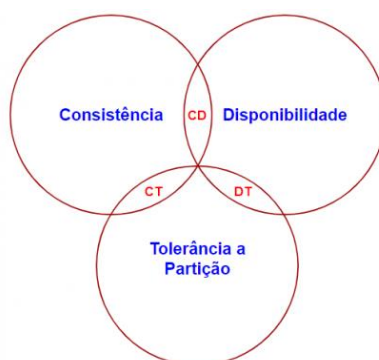
distribuída, os outros nós poderão responder às solicitações de consultas de dados dos clientes.

Teorema CAP:

- De acordo com o teorema CAP, um sistema distribuído de bancos de dados somente pode operar com dois desses comportamentos ao mesmo tempo, mas jamais com os três simultaneamente.

Consistência Eventual é um conceito interessante derivado do teorema CAP. O sistema prioriza as escritas de dados (armazenamento), sendo o sincronismo entre os nós do servidor realizado em um momento posterior – o que causa um pequeno intervalo de tempo no qual o sistema como um todo é inconsistente. Para isso, são implementadas as propriedades Disponibilidade e Tolerância a Partição.

Figura 1 – Teorema CAP.



Exemplos de sistemas de bancos de dados que implementam a consistência eventual são o MongoDB, Cassandra e RavenDB (bancos NoSQL) entre outros.

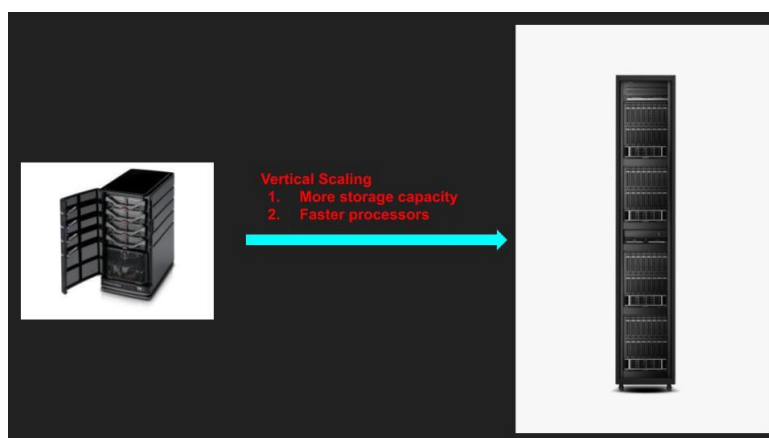
Em Bancos Relacionais é muito comum implementar as propriedades Consistência e Disponibilidade. Como exemplos, citamos os SGBDRs Oracle, MySQL, PostgreSQL, SQL Server e outros.

Ao criar um banco de dados distribuído é importante ter em mente o teorema CAP. Você terá de decidir se o banco será consistente ou disponível, pois bancos de dados distribuídos são sempre tolerantes a partição.

Escalando um banco de dados relacional

Vamos imaginar que começamos um negócio na Internet em meados dos anos 2000. Nosso negócio demonstra um crescimento exponencial no tráfego do site. Nossos clientes reclamam do carregamento lento das páginas da web. O que faremos a seguir? Pedimos aos nossos DBAs para otimizar as consultas de banco de dados e usar índices para melhorar o desempenho do site. Poucos meses depois, começamos a receber reclamações novamente. A escala vertical agora vem em nosso auxílio. Investir valores a mais na compra de servidores maiores resolve nosso problema.

Figura 2 – Escalonamento



Existem limites reais sobre o quão longe podemos ir com a escala vertical de nossos bancos de dados. E se nosso site quiser entrar em um novo negócio e armazenar vídeos, imagens, chats e todas as formas de outros dados? A resposta é simples. Como uma única máquina pode armazenar uma quantidade limitada de dados, temos que recorrer ao escalonamento horizontal. Compramos servidores grandes e distribuímos os dados e o tráfego por essas máquinas. Os bancos de dados SQL não são projetados para escala horizontal. A união de conjuntos de dados e agregação de dados de muitas máquinas introduz complexidade em nosso projeto.

Bancos de dados relacionais tradicionais, como MySQL e PostgreSQL oferecem suporte a transações A.C.I.D. (Atomicidade, Consistência, Isolamento e Durabilidade). Os bancos de dados No-SQL são compatíveis com B.A.S.E.

(Basicamente disponível eventualmente consistente). Vamos fazer um tour pelos recursos dos bancos de dados NoSQL.

Recursos de bancos de dados NoSQL

Abaixo elencamos alguns dos principais recursos de banco de dados NoSQL.

1 – Sem esquema ou com esquema flexível:

Os bancos de dados relacionais têm um esquema rígido. Os usuários precisam passar por várias iterações para modelar os dados. Alterar o tipo de dados de um atributo se torna um pesadelo para desenvolvedores, clientes potenciais e DBAs. O NoSQL supera essa limitação fornecendo um esquema flexível. Esses bancos de dados abstraem o armazenamento de dados e o trabalho interno dos usuários. Eles fornecem suporte para armazenar estruturas de dados definidas pelo usuário. Por exemplo: os dados podem ser armazenados na forma de um objeto JSON. Os usuários têm flexibilidade para adicionar, substituir ou remover atributos dos dados.

Por serem agnósticos quanto ao esquema, os bancos de dados NoSQL também são denominados bancos de dados schema-on-read. Você só precisa saber como os dados são armazenados durante a leitura dos dados.

O esquema flexível encurta o tempo de desenvolvimento. Você não precisa mais passar por muitas iterações de modelagem e design de dados. Os desenvolvedores podem armazenar e recuperar o que quiserem. A única desvantagem do design sem esquema é que ele aumenta o risco de inconsistência, pois há uma falta de controle.

Os bancos de dados SQL oferecem suporte a valores nulos para colunas. Por exemplo: a página da web de um aplicativo de banco tem muitos campos opcionais como nome da rua. Se os usuários não preencherem os campos opcionais, o banco de dados ainda reservará espaço para essas colunas, caso os usuários os atualizem

no futuro. Em bancos de dados NoSQL, você não passa as entradas nulas e, portanto, o armazenamento é otimizado.

Sem esquema não significa que qualquer lixo aleatório pode ser armazenado no banco de dados. Por exemplo, se uma coluna de banco de dados suporta o tipo de dados JSON, o JSON deve ser bem formatado. O aplicativo obterá um erro se tentar armazenar um objeto JSON malformatado.

2 - Não relacional:

Bancos de dados relacionais organizam dados em linhas e colunas. Você pode armazenar dados em muitas tabelas e as tabelas podem ter relacionamentos diferentes. Para buscar os dados você pode juntar as tabelas no valor de um atributo. O desempenho do aplicativo diminui quando o número de tabelas a serem unidas chega a dois dígitos ou mais. Há uma queda significativa na velocidade no caso de o aplicativo unir tabelas armazenadas em servidores de banco de dados diferentes.

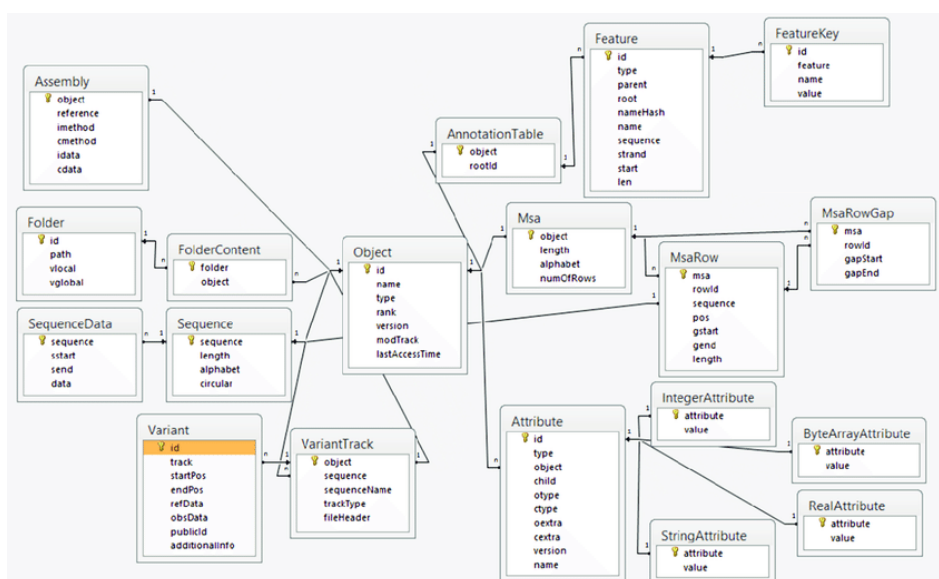
Os bancos de dados NoSQL são desnormalizados. Não existe um conceito de relacionamento entre registros em bancos de dados NoSQL. Você pode armazenar os dados agregados em uma única tabela, em vez de espalhá-los por diferentes tabelas.

A seguir estão as principais vantagens da abordagem acima:

- Velocidade de consulta – A velocidade aumenta significativamente, pois apenas uma pesquisa no atributo-chave é necessária e não há necessidade de juntar muitas tabelas.
- Armazenamento e recuperação – Basta salvar e obter um único registro.

Por exemplo, ao projetar um aplicativo de entrega de comida usando RDBMS, você criará várias tabelas; para usuários, restaurante, pedidos. Em um banco de dados NoSQL, uma única tabela de pedidos pode ter um restaurante e os dados do usuário duplicados em várias linhas. A desvantagem da duplicação de dados é superada pelos benefícios mencionados acima.

Figura 3 – Diagrama.



Você pode evitar criar diagramas ER complexos e escrever consultas SQL complicadas. Com os bancos de dados NoSQL, você pode acelerar seu desenvolvimento e se concentrar em fazer as coisas.

3 – Alta escalabilidade e disponibilidade:

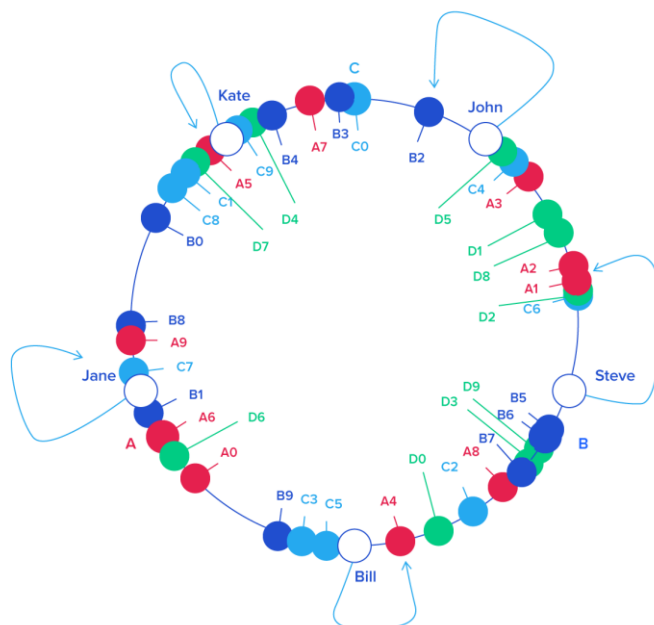
Quando o Google publicou seu artigo sobre o BigTable, ele definiu o BigTable como “um sistema de armazenamento distribuído para gerenciar dados estruturados, que é projetado para escalar para um tamanho muito grande”. O banco de dados NoSQL pode armazenar petabytes de dados em muitos computadores.

Com o aumento exponencial na quantidade de dados não estruturados, ficou difícil armazenar dados em uma única máquina. Os bancos de dados relacionais precisam de hardware especializado para atender à carga sem comprometer o desempenho. Portanto, para escalar tornou-se essencial projetar um sistema que armazenaria dados em um cluster de computadores e os recuperaria de forma eficiente.

Os bancos de dados NoSQL usam servidores comuns que são mais baratos do que servidores de alto desempenho. Conforme o requisito de armazenamento de dados aumenta, mais servidores de commodities podem ser adicionados. Os bancos

de dados NoSQL distribuem os dados uniformemente em um cluster de servidores usando o algoritmo de hash consistente.

Figura 4 – Cluster.



Os bancos de dados NoSQL podem replicar os dados em muitas máquinas. Os dados ainda podem estar acessíveis se qualquer um dos servidores morrer ou travar. Portanto, os bancos de dados NoSQL estão altamente disponíveis.

4 – Código aberto:

O desenvolvimento de código aberto torna o software NoSQL único. Poucos fornecedores de código aberto lançam um produto de código aberto e vendem recursos complementares corporativos. Essas empresas têm um modelo de negócios semelhante ao da RedHat.

A seguir está uma lista de alguns bancos de dados NoSQL de código aberto:

- MongoDB.
- Apache Cassandra.

- Redis.
- Voldemort.
- HyperTable.
- Neo4j.

Equívocos comuns de bancos de dados NoSQL

1 – NoSQL é um único tipo de banco de dados:

Os bancos de dados NoSQL são classificados quanto ao tipo de dados e seu funcionamento interno. A seguir estão os diferentes tipos de bancos de dados NoSQL:

- Chave-valor: esses bancos de dados funcionam como HashMap e podem armazenar qualquer tipo de valor. Alguns exemplos são Redis, Voldemort e Aerospike.
- Colunas: os nomes e o formato das colunas podem variar nas linhas. Cassandra, BigTable e Hypertable são amplos armazenamentos de colunas.
- Documentos: bancos de dados como CouchDB, MongoDB e DocumentDB são capazes de armazenar dados na forma de documentos JSON, XML.
- Grafos: bancos de dados como Neo4j, entidades de modelo interno, nós de gráfico e os relacionamentos entre entidades são indicados por arestas entre os nós.

2 – Risco de perda de dados usando NoSQL:

Uma vez que os bancos de dados NoSQL comprometem a consistência ao invés da disponibilidade, pode haver casos em que cada leitura não segue a gravação mais recente. No entanto, esses bancos de dados são eventualmente consistentes e, portanto, garantem a durabilidade dos dados.

3 – NoSQL é apenas uma palavra da moda:

Amazon, Google, Microsoft, IBM, Oracle e muitas outras grandes corporações construíram bancos de dados NoSQL e estão alavancando seus recursos em sistemas de produção. Grandes empresas de software só investem em tecnologias se tiverem lucro, portanto o NoSQL não é mais um exagero.

4 – RDBMS aprimorado irá substituir NoSQL:

Recursos altamente distribuídos do NoSQL estão sendo integrados à tecnologia RDBMS, o que resultou no surgimento de muitos bancos de dados NewSQL. Os bancos de dados NewSQL superaram a maioria das críticas relacionadas à tecnologia RDBMS. No entanto, os bancos de dados NoSQL são construídos para resolver diferentes problemas de dados usando diferentes estruturas de dados.

Portanto, NoSQL é um termo genérico que define bancos de dados não-relacionais. Não quer dizer que seus modelos não possuem relacionamentos e sim que não são orientados a tabelas. Not Only SQL (não apenas SQL).

Bancos de dados NoSQL são cada vez mais usados em Big Data e aplicações web de tempo real.

O NoSQL fornece esquemas flexíveis e escala facilmente com grandes quantidades de dados e altas cargas de usuário, assim como bancos de dados flexíveis, escaláveis, de alto desempenho e altamente funcionais para fornecer excelentes experiências ao usuário.

NoSQL é um DBMS não relacional, que não requer um esquema fixo, evita junções e é fácil de escalar. O objetivo de usar um banco de dados NoSQL é para armazenamentos de dados distribuídos com enormes necessidades de armazenamento de dados. NoSQL é usado para Big Data e aplicativos da web em tempo real. Por exemplo, empresas como Twitter, Facebook e Google coletam terabytes de dados do usuário todos os dias.

O RDBMS tradicional usa a sintaxe SQL para armazenar e recuperar dados para novas percepções. Em vez disso, um sistema de banco de dados NoSQL abrange uma ampla gama de tecnologias de banco de dados que podem armazenar dados estruturados, semiestruturados, não estruturados e polimórficos.

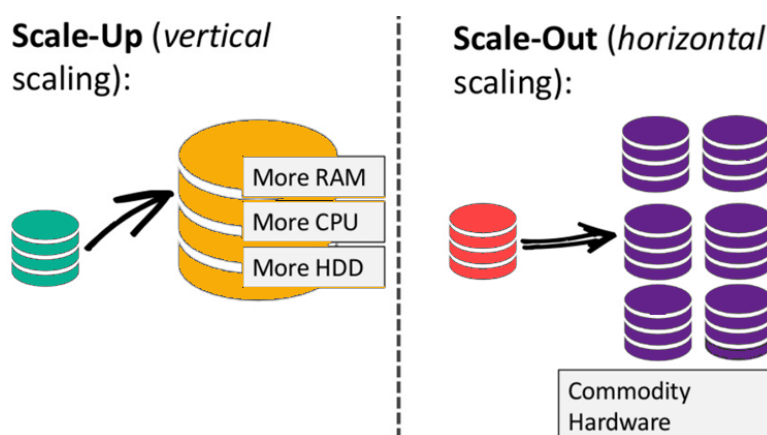
Por que usar o NoSQL?

O conceito de bancos de dados NoSQL se tornou popular entre gigantes da Internet como Google, Facebook, Amazon etc., que lidam com grandes volumes de dados. O tempo de resposta do sistema torna-se lento quando você usa RDBMS para grandes volumes de dados.

Para resolver esse problema, poderíamos “aumentar” nossos sistemas atualizando nosso hardware existente. Esse processo é caro.

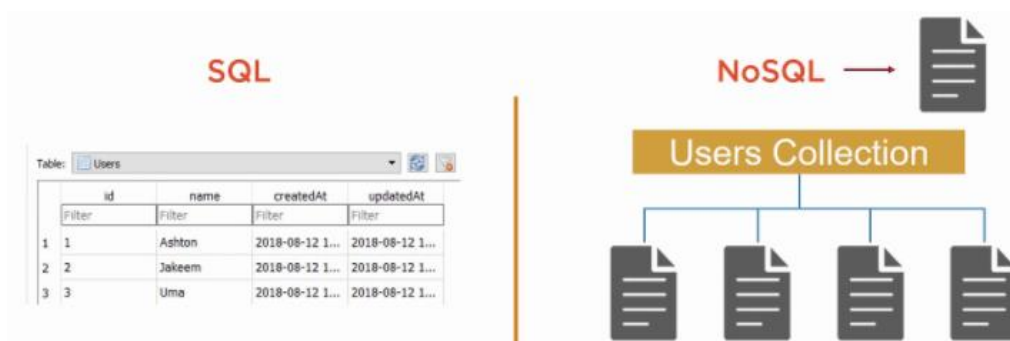
A alternativa para esse problema é distribuir a carga do banco de dados em vários hosts sempre que a carga aumentar. Este método é conhecido como “escalonamento”.

Figura 5 – Escalonamento Vertical e Horizontal.



O banco de dados NoSQL não é relacional, portanto é melhor escalonável do que os bancos de dados relacionais, pois são projetados com aplicativos da web em mente.

Figura 61 – Banco NoSQL.



Propriedades dos Bancos de Dados

Os Bancos de Dados Relacionais são sistemas eficientes, o que os torna uma escolha comum para armazenar registros financeiros, informações logísticas, dados pessoais e outras informações em novas bases de dados. Por serem mais fáceis de entender e usar do que os bancos de dados NoSQL, os bancos de dados relacionais também frequentemente substituem bancos de dados hierárquicos legados e bancos de dados de rede.

Os Bancos de Dados Relacionais têm as seguintes propriedades:

- Os valores são atômicos.
- Todos os valores em uma coluna têm o mesmo tipo de dados.
- Cada linha é única.
- A sequência de colunas é insignificante.
- A sequência de linhas é insignificante.
- Cada coluna tem um nome único.
- As restrições de integridade mantêm a consistência dos dados em várias tabelas.

O NoSQL é uma alternativa sem esquema para SQL e RDBMSs projetados para armazenar, processar e analisar quantidades extremamente grandes de dados não estruturados.

Nas bases de dados NoSQL, os princípios do ACID (atomicidade, consistência, isolamento e durabilidade) são reduzidos. Além disso, o processo de normalização não é obrigatório no NoSQL. Devido ao tamanho e velocidade dos dados modernos, é preferível que os bancos de dados NoSQL sejam desnormalizados.

Os bancos de dados NoSQL têm as seguintes propriedades:

- Maior escalabilidade.
- Usam computação distribuída.
- São econômicos.
- Suportam esquema flexível.
- São capazes de processar dados não estruturados e semiestruturados.
- Não há relações complexas, como as entre mesas em um RDBMS.

A tabela a seguir mostra os tipos de bancos de dados não relacionais e os recursos associados a eles:

Tipo	Desempenho	Escalabilidade	Flexibilidade	Complexidade
Chave-valor	Alto	Alto	Alto	Alto
Colunar	Alto	Alto	Moderado	Baixo
Documento	Alto	Variável para alto	Alto	Baixo
Grafo	Variável	Variável	Alto	Alto

Visão geral de NewSQL

NewSQL é uma classe de sistemas gerenciadores de banco de dados relacionais que buscam fornecer a escalabilidade dos sistemas NoSQL, para cargas

de trabalho de processamento de transações on-line (OLTP), mantendo as garantias ACID de um sistema de banco de dados tradicional.

Muitos sistemas corporativos que lidam com dados de alto perfil (por exemplo, sistemas financeiros e de processamento de pedidos) são muito grandes para bancos de dados relacionais convencionais, mas têm requisitos transacionais e de consistência que não são práticos para sistemas NoSQL. As únicas opções disponíveis anteriormente para essas organizações eram comprar computadores mais poderosos ou desenvolver middleware personalizado que distribua solicitações sobre DBMS convencionais. Ambas as abordagens apresentam altos custos de infraestrutura e/ou custos de desenvolvimento. Os sistemas NewSQL tentam conciliar os conflitos.

O termo foi usado pela primeira vez pelo Group Matthew Aslett em um artigo de pesquisa de 2011, que discutiu o surgimento de uma nova geração de sistemas de gerenciamento de banco de dados. Um dos primeiros sistemas NewSQL foi o sistema de banco de dados paralelo H-Store.

Figura 7 – New SQL.



Aplicações típicas são caracterizadas por grandes volumes de transações OLTP:

- São de curta duração.
- Trabalha com pequenas quantidades de dados por transação.
- Usa pesquisas indexadas (sem varreduras de tabela).

- Têm um pequeno número de formulários (um pequeno número de consultas com diferentes argumentos).

No entanto, alguns suportam aplicações de processamento transacional/analítica híbrido (HTAP). Tais sistemas melhoram o desempenho e a escalabilidade, omitindo recuperação de pesos pesados ou controle de concorrência.

As duas características de distinção comuns das soluções de banco de dados NewSQL, são as que suportam a escalabilidade on-line dos bancos de dados NoSQL e o modelo de dados relacionais (incluindo a consistência ACID) usando SQL como sua interface primária.

Os sistemas NewSQL podem ser agrupados em três categorias:

- Novas arquiteturas: os sistemas NewSQL adotam várias arquiteturas internas. Alguns sistemas empregam um conjunto de nós compartilhado, no qual cada nó gerencia um subconjunto dos dados. Eles incluem componentes como controle de concorrência distribuída, controle de fluxo e processamento de consulta distribuído.
- Motores SQL: a segunda categoria são motores de armazenamento otimizados para SQL. Esses sistemas fornecem a mesma interface de programação do SQL, mas dimensionam melhor do que os motores embutidos.
- Fragmento transparente: esses sistemas dividem automaticamente os bancos de dados em vários, usando o algoritmo de consenso Raft ou Paxos.

Recursos e técnicas em bancos NoSQL

Temos como recursos nos bancos de dados NoSQL:

1. Não relacional:

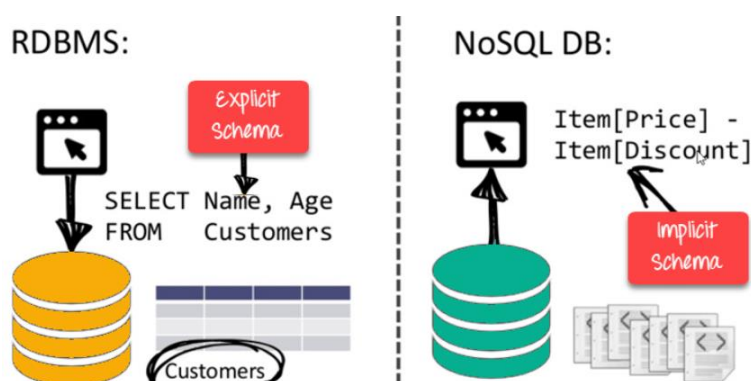
- Os bancos de dados NoSQL não seguem o modelo relacional.

- Não usa tabelas com registros de coluna fixa simples.
- Dados podem ser agregados, autocontidos ou BLOBs.
- Não requer mapeamento relacional de objeto e normalização de dados.
- Sem recursos complexos, como linguagens de consulta, planejadores de consulta, junções de integridade referencial e ACID.

2. Esquema-livre ou flexível:

- Os bancos de dados NoSQL são livres de esquema ou têm esquemas flexíveis.
- Não requer nenhum tipo de definição do esquema dos dados.
- Oferece estruturas heterogêneas de dados no mesmo domínio.

Figura 8 – Esquema NoSQL.



3. API simples:

- Oferecem interfaces fáceis de usar para armazenamento e consulta de dados fornecidos.
- APIs permitem manipulação de dados de baixo nível e métodos de seleção.
- Protocolos baseados em texto, usando principalmente HTTP REST com JSON.
- Geralmente não usam linguagem de consulta padrão.

- Bancos de dados habilitados para web, funcionando como serviços voltados para a internet.

4. Distribuído:

- Vários bancos de dados NoSQL podem ser executados de forma distribuída.
- Oferecem recursos de escalonamento automático e failover.
- Muitas vezes, o conceito de ACID pode ser sacrificado para manter a escalabilidade e performance.
- Quase sempre sem replicação síncrona entre nós distribuídos.
- Replicação multimestre assíncrona, ponto a ponto e Replicação HDFS.
- Fornecendo apenas consistência eventual.
- Arquitetura sem muito compartilhamento. Isso permite menos coordenação e maior distribuição.

Sobre as principais características nos bancos de dados NoSQL, é importante ressaltar algumas técnicas utilizadas para a implementação de suas funcionalidades. Entre elas estão:

- Map/reduce.
- Consistent hashing.
- MVCC - Multiversion Concurrency Control.
- Vector Clocks.

Map Reduce: permite a manipulação de enormes volumes de dados ao longo de nós em uma rede.

Funciona da seguinte forma: na fase MAP os problemas são particionados em pequenos problemas, que são distribuídos em outros nós na rede.

Quando chegam à fase REDUCE, esses pequenos problemas são resolvidos em cada nó filho e o resultado é passado para o pai, que sendo ele consequentemente filho, repassaria para o seu, até chegar à raiz do problema.

Hashing: é o processo de mapear um dado, normalmente um objeto de tamanho arbitrário para outro dado de tamanho fixo, sendo geralmente um inteiro, conhecido como código hash ou simplesmente hash. Uma função que é geralmente usada para mapear objetos para código hash, conhecido como função hash.

Por exemplo, uma função hash pode ser usada para mapear strings de tamanho aleatório para algum número fixo entre 0... N. Dado qualquer string, ela sempre tentará mapeá-la para qualquer número inteiro entre 0 e N. Suponha que N seja 100. Então, por exemplo, para qualquer string, a função hash sempre retornará um valor entre 0 e 100.

Hello ---> 60

Hello World ---> 40

Hashing consistente é um tipo especial de hashing, que quando uma tabela hash é redimensionada, apenas as chaves precisam ser remapeadas em média, onde é o número de chaves e é o número de slots.

Suporta mecanismos de armazenamento e recuperação, onde a quantidade de sites está em constante mudança. É interessante usar essa técnica, pois ela evita que haja uma grande migração de dados entre estes sites, que podem ser alocados ou desalocados para a distribuição dos dados.

Distributed Hashing: suponha que um número de funcionários continue crescendo e se torne difícil armazenar todas as informações dos funcionários em uma tabela hash, que pode caber em um único computador. Nessa situação, tentaremos distribuir a tabela de hash para vários servidores para evitar a limitação de memória de um servidor. Os objetos (e suas chaves) são distribuídos entre vários servidores.

Este tipo de configuração é muito comum para caches na memória como Memcached, Redis etc.

NAME	AGE	CAR	GENDER
jim	36	camaro	M
carol	37	345s	F
johnny	12	supra	M
suzy	10	mustang	F

PARTITION KEY	MURMUR3 HASH VALUE
jim	-2245462676723223822
carol	7723358927203680754
johnny	-6723372854036780875
suzy	1168604627387940318

MVCC – Multiversion Concurrency Control: oferece suporte a transações paralelas em banco de dados. Por não fazer uso de locks para controle de concorrência, faz com que transações de escrita e leitura sejam feitas simultaneamente.

Ao serem iniciados novos processos de leitura de um banco de dados, e nesse mesmo instante existir um outro processo que está atualizando, pode acontecer que o processo de leitura esteja lendo veja apenas uma parte do que está sendo atualizado, ou seja, um dado inconsistente.

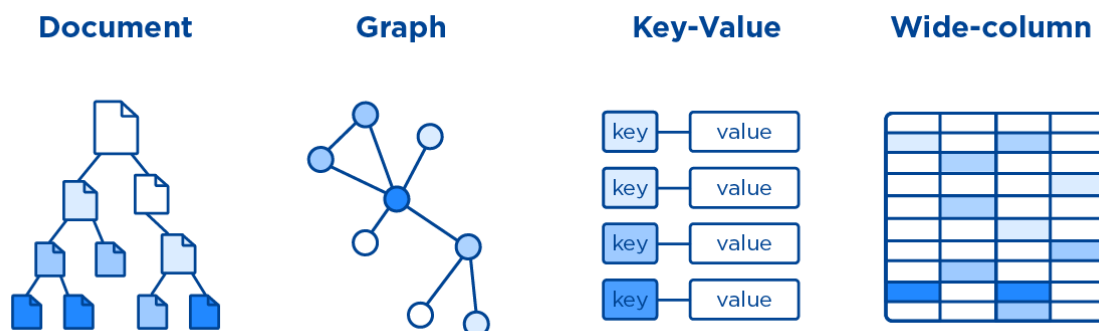
Vector clocks: ordenam eventos que ocorreram em um sistema. Como existe a possibilidade de várias operações estarem acontecendo simultaneamente, o uso de um log de operações informando suas datas se faz importante para informar qual versão de um dado é a mais atual.

Tipos de bancos NoSQL

Existem quatro grandes tipos de NoSQL: armazenamento de valor-chave, armazenamento de documento, banco de dados orientado a colunas e banco de

dados gráfico. Cada tipo resolve um problema que não pode ser resolvido com bancos de dados relacionais.

Figura 92 – Tipos de NoSQL.



Existem quatro grandes tipos de NoSQL:

- Armazenamento de chave-valor.
- Armazenamento de documentos.
- Banco de dados orientado a colunas.
- Banco de dados de gráficos.

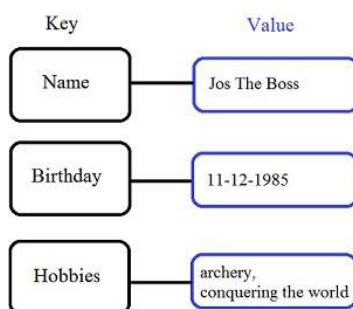
Cada tipo resolve um problema que não pode ser resolvido com bancos de dados relacionais.

NoSQL Chave-valor

Os dados são armazenados em pares chave/valor. Ele foi projetado para lidar com muitos dados e cargas pesadas.

Os bancos de dados de armazenamento de par de chave-valor armazenam dados como uma tabela hash, em que cada chave é única e o valor pode ser JSON, BLOB (objetos binários grandes), string etc.

Figura 10 – Chave/Valor.



O valor em um armazenamento de chaves-valor podem ser qualquer coisa: uma string, um número, mas também um conjunto inteiramente novo de pares de chaves-valor encapsulado em um objeto.

Exemplos de armazenamentos de valores-chave são Redis, Voldemort, Riak e DynamoDB da Amazon.

```
{
  "internal data": [
    {
      "entities": [
        {
          "customer": [
            {
              "id": 1, "name": "Freddy"
            },
            {
              "id": 2, "name": "Fritz"
            }
          ]
        },
        {
          "legal entities": [
            {
              "id": 1, "company": "Maiton"
            }
          ]
        }
      ]
    },
    {
      "Products": [
        {
          "furniture": [
            {
              "id": 1, "name": "Octopus Table", "stock": 1
            }
          ]
        }
      ]
    }
  ]
}
```

NoSQL orientado a documentos

Os armazenamentos de documentos são um passo à frente em complexidade em relação aos armazenamentos de valores-chave: um armazenamento de documentos assume uma certa estrutura de documento que pode ser especificada com um esquema. Os armazenamentos de documentos parecem ser os mais naturais entre os tipos de banco de dados NoSQL, porque são projetados para armazenar documentos do dia a dia como estão e permitem consultas e cálculos complexos

nesta forma de dados, frequentemente já agregada. A maneira como as coisas são armazenadas em um banco de dados relacional, faz sentido do ponto de vista da normalização: tudo deve ser armazenado apenas uma vez e conectado por meio de chaves estrangeiras. Os armazenamentos de documentos se preocupam pouco com a normalização, desde que os dados estejam em uma estrutura que faça sentido. Um modelo de dados relacionais nem sempre se encaixa bem com determinados casos de negócios.

Jornais ou revistas, por exemplo, contêm artigos. Para armazená-los em um banco de dados relacional, você precisa primeiro dividi-los: o texto do artigo vai para uma tabela, o autor e todas as informações sobre o autor em outra, e os comentários sobre o artigo quando publicado em um site vão para outra. Conforme mostrado abaixo, um artigo de jornal também pode ser armazenado como uma entidade única; isso diminui a carga cognitiva de trabalhar com os dados para aqueles acostumados a ver os artigos o tempo todo.

Amazon SimpleDB, CouchDB, MongoDB, Riak e Lotus Notes são sistemas DBMS populares originados de documentos.

NoSQL Colunares

Bancos de dados orientados a colunas funcionam em colunas e são baseados em papel BigTable do Google. Cada coluna é tratada separadamente. Os valores de bancos de dados de coluna única são armazenados de forma contígua.

ColumnFamily			
Row Key	Column Name		
	Key	Key	Key
	Value	Value	Value
	Column Name		
	Key	Key	Key
	Value	Value	Value

Eles oferecem alto desempenho em consultas de agregação, como SUM, COUNT, AVG, MIN etc., pois os dados estão prontamente disponíveis em uma coluna.

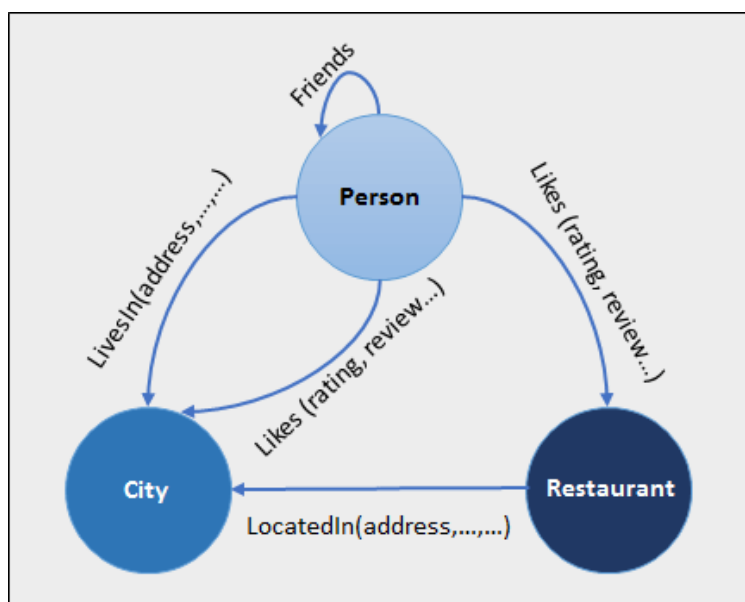
Bancos de dados NoSQL baseados em colunas, são amplamente usados para gerenciar Data Warehouses, Business Intelligence, CRM e catálogos de cartões de biblioteca.

HBase, Cassandra, HBase e Hypertable são exemplos de um banco de dados baseado em colunas.

NoSQL Grafos

Um banco de dados do tipo grafo armazena entidades, bem como as relações entre essas entidades. A entidade é armazenada como um nó com o relacionamento como arestas. Uma aresta fornece um relacionamento entre os nós. Cada nó e borda possui um identificador exclusivo.

Figura 11 – NoSQL Grafos.



Comparado a um banco de dados relacional em que as tabelas são fracamente conectadas, um banco de dados grafo é multi-relacional por natureza. Atravessar relacionamentos tão rápido quanto já foram capturados no banco de dados e não há necessidade de calculá-los.

Bancos de dados de base de grafos são usados principalmente para redes sociais, logística, dados espaciais.

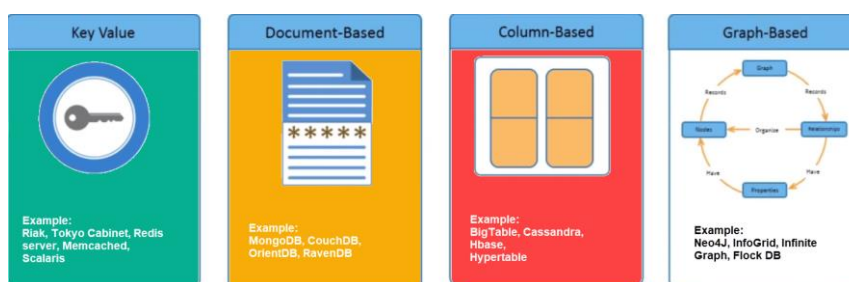
Neo4J, Infinite Graph, OrientDB e FlockDB são alguns bancos de dados populares baseados em gráficos.

Ferramentas de mecanismo de consulta para NoSQL

O mecanismo de recuperação de dados mais comum é a recuperação baseada em REST de um valor com base em sua chave / ID com recurso GET.

Banco de dados de armazenamento de documentos oferecem consultas mais difíceis, pois eles entendem o valor em um par de chave-valor. Por exemplo, CouchDB permite definir visualizações com MapReduce.

Figura 3 – Ferramentas NoSQL



Motivações no uso do NoSQL

Podemos citar as vantagens e desvantagens no uso de banco de dados NoSQL, o que seria uma boa visão para entender o que seriam as motivações para o seu uso.

Vantagens do NoSQL:

- Pode ser usado como fonte de dados primária ou analítica.
- Capacidade de Big Data.
- Nenhum ponto único de falha.
- Replicação fácil.
- Não há necessidade da camada de cache separada.
- Ele fornece desempenho rápido e escalabilidade horizontal.
- Pode lidar com dados estruturados, semiestruturados e não estruturados com o mesmo efeito.
- Programação orientada a objetos, fácil de usar e flexível.
- Os bancos de dados NoSQL não precisam de um servidor dedicado de alto desempenho.
- Suporte aos principais idiomas e plataformas do desenvolvedor.
- Mais simples de implementar do que usar RDBMS.
- Ele pode servir como fonte de dados primário para aplicativos online.
- Lida com Big Data que gerencia a velocidade, variedade, volume e complexidade dos dados.
- Excelente em banco de dados distribuído e operações de multi-data center.
- Elimina a necessidade de uma camada de cache específica para armazenar dados.
- Oferece um design de esquema flexível que pode ser facilmente alterado sem tempo de inatividade ou interrupção do serviço.

Desvantagens:

- Sem regras de padronização.
- Recursos de consulta limitados.
- Bancos de dados e ferramentas RDBMS são comparativamente maduros.
- Ele não oferece nenhum recurso de banco de dados tradicional, como consistência quando várias transações são realizadas simultaneamente.
- Quando o volume de dados aumenta, é difícil manter valores únicos, pois as chaves se tornam difíceis.
- Não funciona tão bem com dados relacionais.
- A curva de aprendizado é difícil para novos desenvolvedores.
- As opções de código aberto não são tão populares para empresas.

Alguns bancos NoSQL

Aerospike: banco de dados NoSQL que oferece uma vantagem de velocidade de memória, atraindo empresas de anúncios de alta escala e aquelas que precisam de tempos de resposta em milissegundo. Aerospike está apostando em novas categorias, incluindo jogos, e-commerce e segurança, onde a baixa latência é tudo.

Apache Cassandra: os pontos fortes são a modelagem de dados NoSQL e escalabilidade linear flexível em hardware por conta do uso de cluster.

Amazon DynamoDB: foi desenvolvido pela Amazon para incrementar o seu e-commerce, possibilitando ter seus serviços altamente escaláveis. Inspirou o Cassandra, Riak e outros projetos NoSQL.

MongoDB: é o banco de dados mais popular NoSQL, com mais de sete milhões de downloads e centenas de milhares de implantações. Sua popularidade se deve à facilidade de desenvolvimento e manejo flexível dos dados. Muito utilizado em aplicações de redes sociais web e móvel.

HBase: é o banco de dados que roda em cima do HDFS (Hadoop Distributed File System – sistema de arquivos distribuído), por isso dá aos usuários a capacidade única de trabalhar diretamente com os dados armazenados no Hadoop. As características incluem grande escalabilidade.

Redis: é o banco de dados NoSQL do tipo chave-valor mais conhecido. No mercado podemos encontrar diversas outras soluções que também são mecanismos de armazenamento baseado em chave-valor.

Capítulo 2. CRUD no MongoDB

O MongoDB é um banco de dados não-relacional que traz o conceito de Banco de Dados Orientado a Documentos. Ele tem como característica conter informações importantes em um único documento. Deste modo, possibilita a consulta de documentos através de métodos avançados de agrupamento e filtragem.

Ele é um banco de dados NoSQL, onde não há presença de SQL. Deste modo, este tipo de banco de dados não traz consigo os fundamentos de um modelo relacional e com a linguagem SQL. Os bancos de dados orientados a documentos não fornecem relacionamentos entre documentos, o que mantém seu design sem esquemas.

Mongod.exe

Mongod é o processo principal para o sistema MongoDB. Ele lida com solicitações de dados, gerencia o acesso aos dados e executa operações de gerenciamento em segundo plano.

Fica na pasta bin.

--help, -h: retorna as informações e opções de uso do mongod

--version: retorna a versão de uso do mongod

--config <filename>, -f <filename>

Logs

MongoDB possui um recurso sofisticado de criação de perfil. O log acontece na system.profile. Os logs podem ser vistos em ***db.system.profile.find()***

Existem 3 níveis de log (origem):

- **Nível 0** - o criador de perfil está desativado, não coleta nenhum dado. O mongod sempre grava operações mais longas do que o limite slowOpThresholdMs em seu log. Este é o nível padrão do criador de perfil.

- **Nível 1** - coleta dados de criação de perfil apenas para operações lentas. Por padrão, operações lentas são aquelas mais lentas que 100 milissegundos. Você pode modificar o limite para operações "lentas" com a opção de tempo de execução `slowOpThresholdMs` ou o comando `setParameter`. Consulte a seção Especificar o limite para operações lentas para obter mais informações.
- **Nível 2** - coleta dados de criação de perfil para todas as operações do banco de dados.

Para ver em que nível de perfil o banco de dados está sendo executado, use: **`db.getProfilingLevel()`**; Para ver o status: **`db.getProfilingStatus()`**; Para alterar o status da criação de perfil, use o comando **`db.setProfilingLevel(level, milliseconds)`** onde `level` refere-se ao nível de criação de perfil e `milliseconds` é o ms de qual duração as consultas precisam ser registradas. Para desativar o log, use **`db.setProfilingLevel(0)`**

A consulta para procurar na coleção de perfis do sistema todas as consultas que levaram mais de um segundo, ordenadas por carimbo de data e hora decrescente, será **`db.system.profile.find({ millis : { $gt:1000 } }).sort({ ts : -1 })`**

Collections

Tipos de coleções (collections):

- Tipo expansível: Quanto mais dados adicionados, maior será a coleção.
- Tipo capped (coleção limitada): Podem conter determinadas quantidades de dados até que o documento antigo ser substituído pelo mais novo.

Algumas características das collections:

- O nome da collection tem que ser único.
- O nome pode conter letras e números.
- O símbolo \$ é reservado ao MongoDB.

- Caracterer nulo não é permitido no nome.
- Não poderá começar a string “system”.

Tipos de dados possíveis

MongoDB suporta vários tipos de dados.

- **String**: este é o tipo de dados mais comumente usado para armazenar dados. String no mongodb deve ser um UTF-8 válido.
- **Integer**: este tipo é usado para armazenar um valor numérico. Integer pode ser 32 bits ou 64 bits dependendo do seu servidor.
- **Boolean**: este tipo é usado para armazenar um valor booleano (verdadeiro/falso).
- **Double**: este tipo é usado para armazenar valores de ponto flutuante.
- **MinKey / Maxkey**: este tipo é usado para comparar um valor contra os elementos mais baixos e mais altos de um BSON. são usados em operações de comparação e existem principalmente para uso interno. Para todos os valores possíveis do elemento BSON, MinKey sempre será o menor valor, enquanto MaxKey sempre será o maior valor.
- **Arrays**: este tipo é usado para armazenar arrays, listas ou múltiplos valores dentro de uma chave(key).
- **Timestamp**: ctimestamp. Isto pode ser útil para a gravação de quando um documento foi modificado ou acrescentado.
- **Object**: este tipo de dado é usado para incorporar documentos.
- **Null**: este tipo é usado para armazenar um valor nulo(null).

- **Symbol:** este tipo de dado é usado de forma idêntica ao String, porém ele é geralmente reservado para linguagens que usam um tipo de símbolo específico.
- **Date:** este tipo de dado é utilizado para armazenar a data ou a hora atual no formato de UNIX. Você pode especificar o seu próprio date_time através da criação do objeto Date, e passando o dia, mês e ano para ele.

Para trabalhar com a função **ISODate()**, devemos especificar a data no formato “YYYY-MM-DD hh:mm:ss”, dentre outros possíveis. Por exemplo: a inserção de um documento com a data de 1º de maio de 2012 às 12h30 ficaria:
db.ColDate.insert({ aDate: new ISODate('2012-05-01 12:30:00') })

Para obter partes da data, podemos utilizar os seguintes métodos do tipo de dados Date:

- getDate() obtém o dia do mês: > doc.aDate.getDate().
- getMonth() obtém o mês - Janeiro=0 e Dezembro=11: > doc.aDate.getMonth().
- getFullYear() obtém a ano: > doc.aDate.getFullYear().
- **Object ID:** este tipo de dados é usado para armazenar os identificadores(ID) dos documentos.
- **Binary data:** este tipo de dados é usado para armazenar um dado binário.
- **Code:** este tipo de dados é usado para armazenar código javascript dentro do documento.
- **Regular expression:** este tipo de dados é usado para armazenar expressões regulares.

ObjectId

Em bancos de dados relacionais, temos as primary Keys, que são chaves primárias. É um identificador único da linha da tabela. No MongoDB temos o ObjectId.

O valor ObjectId hexadecimal de 12 bytes consiste em:

- Um valor de timestamp (carimbo pela data/hora) de 4 bytes, representando a criação do ObjectId, medido em segundos.
- Um valor aleatório de 5 bytes (elimina conflitos em casos de inserção de registro no mesmo instante).
- Um contador de incremento de 3 bytes, inicializado com um valor aleatório.
- **ObjectId.getTimestamp ()**: retorna a parte do carimbo de data / hora do objeto como uma Data.
- **ObjectId.toString ()**: retorna a representação JavaScript na forma de um literal de string "ObjectId (...)".
- **ObjectId.valueOf ()**: retorna a representação do objeto como uma string hexadecimal. A string retornada é o atributo str.

Observação: o ObjectId usa da data do cliente e não do server. Se o cliente estiver com datas erradas, você refletirá isso no ObjectId.

CRUD

O CRUD é um acrônimo para Create, Read, Update e Delete. São as 4 operações principais em um banco de dados. No MongoDB essas funcionalidades são:

- Create - insert(); insertOne(); insertMany().
- Read - find(); findOne().
- Update – updateOne(); updateMany(); replaceOne().

- Delete - `deleteOne()`; `deleteMany()`.

Create:

- `db.collection.insert(<document>)`.
- `db.collection.save(<document>)`.
- `db.collection.update(<query>, <update>, { upsert: true })`.

A partir da versão 3.2 do MongoDB:

- `db.collection.insertOne(<document>,{writeConcern: <document>})`
- `db.collection.insertMany([<document 1>, <document 2>, ...], {writeConcern: <document>, ordered: <boolean>})`

Read:

- `db.collection.find(<query>, <projection>)`.
- `db.collection.findOne(<query>, <projection>)`.

Update:

- `db.collection.updateOne(<query>, <update>, <options>)`.
- `db.collection.updateMany({nome: /a/}, {$set:{salario: 2000}})`.
- `db.collection.replaceOne({nome: /Perla/}, {salario: 5000})`.

Delete:

- `db.collection.deleteOne(filter)`

Insert

O comando insert como o próprio nome já diz insere um documento na collection referenciada.

```
db.teste.insert({nome: "Usuario 1", idade: 25, data_nascimento: "01/03/1990"})
```

Assim, inserimos na collection teste um novo usuário, chamado Usuario 1, com idade = 25 e nascido no dia 01/03/1990.

A inserção se dá pelo método com a sintaxe db.teste.insert(), onde o db é o banco de dados em que estamos no momento setado pelo use, o teste é o nome da collection em seguida insert que nada mais é do que o nome do método que por sinal recebe um objeto JSON como parâmetro contendo os dados a serem inseridos no banco. Essa forma singular de dados é o que chamamos de *document*.

Da mesma maneira que o use, quando passamos dentro do método de inserção o nome da collection, se ele não existir ela será criada pelo Mongo Shell.

Além do método db.collection.insert(), temos também os métodos db.collection.insertOne() e db.collection.insertMany(). O primeiro é utilizado para inserção de apenas um documento, e ao invés de retornar um objeto WriteResult, o resultado é um objeto contendo o status de sucesso da operação, assim como o ObjectID gerado na inserção.

```
db.collection.insertOne(<document>{writeConcern: <document>})
```

WriteConcern - Opcional. Um documento expressando a preocupação de gravação. Omita o uso da preocupação de gravação padrão, não defina explicitamente a preocupação de gravação para a operação se executado em uma transação. Retorna um booleano como verdadeiro se a operação foi executada com preocupação de gravação, ou falso, se a preocupação com gravação foi desabilitada.

O objeto passado como argumento possui duas chaves: w e j. A opção w define a necessidade de um cliente em garantir a escrita de dados em um, nenhum,

ou mais de um servidor mongodb. Para entender de fato o funcionamento dessa opção, é necessário entender um pouco sobre Replica Sets.

De forma resumida:

- A opção {w: 1} (padrão no mongodb) exige que a informação seja escrita em uma instância do mongodb para retornar um código de sucesso ao cliente inserindo o dado.
- A opção {w: 0} exibe o código de sucesso sem a garantia da persistência de dados.
- Já a opção {w: "Majority"} faz com que os dados sejam inseridos na maioria dos membros de um Replica Set (conjunto de servidores mongodb, com o intuito de replicar dados e fornecer alta-disponibilidade).
- A opção j define a necessidade de um cliente em garantir a escrita de dados em disco. Caso você não especifique opção j, os dados escritos permanecem em memória RAM, até que eventualmente o mongoDB grave os dados em seu log de transação, chamado oplog. Para garantir a persistência dos dados de forma persistente, passe como argumento a opção {j: True} a cada operação de escrita.

A partir da versão 3.2 do MongoDB: **db.crud.insertOne({field_A: 2123}):**

Já o método db.collection.insertMany(), insere mais de um documento e retorna como resposta ao cliente um documento contendo os ObjectIDs gerados na inserção de cada documento.

A partir da versão 3.2 do MongoDB: **insertMany ([{ }, { }, { }]):**

db.crud.insertOne({field_A: 2123}, {field_B: 2223})

db.collection.insertMany([<document 1> , <document 2>, ...],{writeConcern: <document>, ordered: <boolean>})

Ordered é um booleano opcional. Especifica se a instância do mongod deve executar uma inserção ordenadamente ou não ordenadamente. O padrão é true.

Find

Com o comando find(), podemos verificar a existência da coleção e seus dados. Funciona de maneira semelhante ao select da linguagem SQL:

```
db.teste.find()
```

E o resultado será: { "_id" : ObjectId("5e6a69e61f94c569504d99e4"), "nome": "Usuario 1", "idade" : 25, "data_nascimento" : "01/03/1990" }.

Observação: quando não especificamos um ID, o próprio MongoDB se encarrega da criação de um.

```
db.collection.find(query, projection)
```

Query – opcional: especifica o filtro de seleção usando operadores de consulta. Para retornar todos os documentos em uma coleção, omite este parâmetro ou passe um documento vazio ({}).

Projeção – opcional: especifica os campos a serem retornados nos documentos que correspondem ao filtro da consulta. Para retornar todos os campos nos documentos correspondentes, omite este parâmetro.

Update

O MongoDB possui três métodos para atualização de dados em um documento.

Os métodos updateOne() e updateMany() localizam o documento segundo os critérios especificados e fazem as alterações descritas. A diferença entre eles é a

quantidade de documentos afetadas, enquanto o `updateOne()` afeta apenas um documento, o `updateMany` afeta todos que atendam a determinado critério.

O método `replaceOne()` localiza um único documento que atenda aos critérios especificados e o substitui por um novo documento.

UpdateOne:

O `updateOne` vai te obrigar a usar operadores, ao invés de um documento inteiro para a atualização, o que é muito mais seguro. Sempre que possível, use a chave primária (`_id`) como filtro da atualização, pois ela é sempre única dentro da coleção. Sempre use operadores ao invés de documentos inteiros no segundo parâmetro, independentemente do número de documentos que serão atualizados.

```
db.collection.updateOne(filter, update, options)
```

UpdateMany:

```
db.funcionarios.updateMany({nome: /a/}, {$set:{salario: 2000}})
```

ReplaceOne:

```
db.funcionarios.replaceOne({nome: /Perla/}, {salario: 5000})
```

Delete

O MongoDB possui dois métodos para a remoção de documentos. Os métodos `deleteOne()` e `deleteMany()` localizam o documento segundo os critérios especificados e o removem da base de dados.

O método `deleteOne()` exclui apenas um documento e o método `deleteMany()` exclui vários documentos.

```
> db.collection.deleteOne({"_id":2})
```

```
> db.collection.deleteMany({"nome":"/Direito/"})
```

Operadores

Like:

Em banco de dados relacionais, quando precisamos consultar uma string usamos o like. Em mongo ficará: `db.getCollection('collection').find({"field":/string/});`

Esta consulta é case sensitive, então temos que usar /i no final da consulta `db.getCollection('collection').find({"field":/string/i})`

Sort:

No mongo podemos ordenar os dados da mesma forma que fazemos no banco de dados relacional, para isso usamos o `sort()`.

```
db.getCollection('collection').find({},{"field1":cidade,"field2":estado}).  
sort({"field1":cidade})
```

OR:

O operador `$or` executa uma operação lógica OR em uma matriz de duas ou mais `<expressions>` e seleciona os documentos que satisfazem pelo menos uma das `<expressions>`.

```
{ $or: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }
```

```
db.funcionarios.find ({ $or: [{ 'escolaridade': 'M', 'escolaridade': 'S' } ] }, { _id: false,  
nome: true, endereço: true } )
```

Exemplos de outros operadores do MongoDB

Operadores de comparação e lógicos:

Estrutura de controle que segrega itens de acordo com o operador que está usando. Os operadores de comparação, como o próprio nome já diz, verificam dois

atributos comparando seus valores. Operadores lógicos realizam comparação entre expressões.

Os operadores de comparação são:

- `$eq`: retorna os objetos que têm o valor igual ao especificado. Sintaxe: `{<field> : {$eq: <valor> } }` Exemplo: `{"age" : {$eq : 20} }`
- Ficar escrevendo em uma linha toda a nossa query pode ficar ruim com o tempo. Então vamos criar uma variável para receber nossa query de operação
`>var query = {"age" : {$eq : 20}}`
- `$gt`: retorna os objetos que têm o valor maior ao especificado. Sintaxe: `{<field> : {$gt: <valor> } }` Exemplo: `{"age" : {$gt : 20} }`.
- `$gte`: retorna os objetos que têm o valor maior ou igual ao especificado. Sintaxe: `{<field> : {$gte: <valor> } }` Exemplo: `{"age" : {$gte : 20} }`.
- `$lt`: retorna os objetos que têm o valor menor que o especificado. Sintaxe: `{<field> : {$lt: <valor> } }` Exemplo: `{"age" : {$lt : 20} }`.
- `$lte`: retorna os objetos que têm o valor menor ou igual que o especificado. Sintaxe: `{<field> : {$lte: <valor> } }` Exemplo: `{"age" : {$lte : 20} }`.
- `$ne`: retorna os objetos com valores diferentes do especificado. Sintaxe: `{<field> : {$ne: <valor> } }` Exemplo: `{"age" : {$ne : 28} }`.
- `$in`: retorna os objetos que têm o valor dentre os especificados no array. Sintaxe: `{<field> : {$in: [valor, valor1] } }` Exemplo: `{"age" : {$in : [28,20]} }`.
- `$nin`: retorna os objetos que não têm o valor dentre os especificados no array. Sintaxe: `{<field> : {$nin: [valor, valor1]} }` Exemplo: `{"age" : {$in : [28,20]} }`.

Os operadores lógicos são:

- `$or`: executa a comparação de duas expressões ou mais e retorna os objetos que cumpram com ao menos uma destas. Sintaxe: `{$or : [<expressao1>,`

<expressao2>, <expressao3>]] Exemplo: {\$or : [{"name" : {\$eq : "Gabriel Alves Scavassa"} } , {"name" : {\$eq : "João Paulo"} }]}.

- \$and: executa a comparação de duas expressões ou mais e retorna os objetos que cumpram com todas elas. Sintaxe: {\$and : [<expressao1>, <expressao2>, <expressao3>]} Exemplo: {\$and : [{"name" : {\$eq : "Gabriel Alves Scavassa"} } , {"age" : {\$eq :28 } }]}.
- \$not: retorna os objetos que não compreendem as expressões. Sintaxe: {<field>: { \$not : {<operator> : <valor>} } } Exemplo: {"age": { \$not : {\$gte : 20} } }.
- \$nor: retorna todos os objetos que não estão de acordo com as expressões no array. Sintaxe: { \$nor : [{expressão}, {expressão1}] } Exemplo: {\$nor : [{ "age" : 20}, {"age" : 28}] }.
- \$exists: junta cláusulas e retorna todos os documentos que não estão de acordo. Sintaxe: { <field>: {\$exists : < true || false>}} Exemplo: {city: {\$exists : true}}
- \$text: realiza busca textual no campo especificado. Sintaxe: { \$text: { \$search: "coffee" } } Exemplo: {\$text: { \$search : 'Gabriel' } }.

Para o comando de pesquisa por texto, precisamos criar um index com o campo de texto que faremos a pesquisa. Sintaxe: db.pessoas.createindex({ name : "text"}). Para buscar partes de uma palavra, usaremos o próprio find(). > db.pessoas.find({"name" : /G/}). Fazemos o pipe para separar a letra. Por padrão está como Case Sensitive. > db.pessoas.find({ "name" : /G/i}). Assim tornamos a pesquisa como Case Insensitive.

Agregação

Operações de agregações processam os dados registrados e retornam um resultado agregado. Agregações agrupam valores vindos de múltiplos documentos reunidos, e podem estabelecer uma variedade de operações sobre esta agrupação

de dados para retornar um resultado único. O sql count(*) e group by são equivalentes a uma agregação no MongoDB. Para agregar valores do MongoDB, você precisa usar o método aggregate().

Exemplo: db.funcionarios.aggregate ({\$group: {'_id': '\$salario', 'media': {\$avg:'\$salario'}} }).

Algumas expressões de agregações disponíveis:

- \$sum: soma o valor definido a partir de todos os documentos da coleção. db.mycol.aggregate([{\$group:{_id : "\$by_user", num_tutorial : {\$sum : "\$likes"}}}])
- \$avg: calcula a média de todos os valores dados de todos os documentos da coleção. db.mycol.aggregate([{\$group:{_id : "\$by_user", num_tutorial : {\$avg : "\$likes"}}}])
- \$min: obtém o mínimo dos valores correspondentes de todos os documentos da coleção. db.mycol.aggregate([{\$group:{_id : "\$by_user", num_tutorial : {\$min : "\$likes"}}}])
- \$max: obtém o máximo dos valores correspondentes de todos os documentos da coleção. db.mycol.aggregate([{\$group:{_id : "\$by_user", num_tutorial : {\$max : "\$likes"}}}])
- \$push: insere o valor para um array no documento resultante. db.mycol.aggregate([{\$group:{_id : "\$by_user", url : {\$push: "\$url"}}}])
- \$addToSet: insere o valor para um array no documento resultante mas não faz criações duplicadas. db.mycol.aggregate([{\$group:{_id : "\$by_user", url : {\$addToSet : "\$url"}}}])
- \$first: obtém o primeiro documento vindo de conjunto de documentos da agregação. Tipicamente isto só faz sentido quando vem junto de aplicação prévia de alguma ordenação. "\$sort". db.mycol.aggregate([{\$group:{_id : "\$by_user", first_url : {\$first : "\$url"}}}])

- **\$last**: obtém o último documento vindo de conjunto de documentos da agregação. Tipicamente isso só faz sentido quando vem junto de aplicação prévia de alguma ordenação “\$sort”. `db.mycol.aggregate([{$group:{_id : “$by_user”, last_url : {$last : “$url”}}}]`).
- **\$lookup**: executa uma associação externa à esquerda da outra coleção no mesmo banco de dados, para filtrar documentos da coleção "unida" para processamento. Para cada documento de entrada, o estágio \$ lookup adiciona um novo campo de array cujos elementos são os documentos correspondentes da coleção “unida”.

```
{ $lookup: {  
  from: <collection to join>,  
  localField: <field from the input documents>,  
  foreignField: <field from the documents of the "from" collection>,  
  as: <output array field>}}
```

Capítulo 3. Modelagem e Relacionamentos no MongoDB

Normalização de Dados

O processo de normalização compreende o uso de um conjunto de regras, chamados de formas normais. Ao analisarmos o banco de dados e verificarmos que ele respeita as regras da primeira forma normal, então podemos dizer que o banco está na “primeira forma normal”. Caso o banco respeite as primeiras três regras, então ele está na “terceira forma normal”. Mesmo existindo mais conjuntos de regras para outros níveis de normalização, a terceira forma normal é considerada o nível mínimo necessário para grande parte das aplicações. (Microsoft 2007)

Normalização é o processo de modelar o banco de dados, projetando a forma como as informações serão armazenadas a fim de eliminar, ou pelo menos minimizar, a redundância no banco. Tal procedimento é feito a partir da identificação de uma anomalia em uma relação, decompondo-as em relações melhor estruturadas.

Normalmente precisamos remover uma ou mais colunas da tabela, dependendo da anomalia identificada, e criar uma segunda tabela obviamente com suas próprias chaves primárias, e relacionarmos a primeira com a segunda para assim tentarmos evitar a redundância de informações.

Formas normais:

Primeira Forma Normal: uma relação está na primeira forma normal quando todos os atributos contêm apenas um valor correspondente, singular e não existem grupos de atributos repetidos, ou seja, não admite repetições ou campos que tenham mais que um valor. O procedimento inicial é identificar a chave primária da tabela. Após, devemos reconhecer o grupo repetitivo e removê-lo da entidade. Em seguida, criamos uma nova tabela com a chave primária da tabela anterior e o grupo repetitivo. Por exemplo, um atributo Endereço deve ser subdividido em seus componentes: Logradouro, Número, Complemento, Bairro, Cidade, Estado e CEP.

Segunda Forma Normal: é dito que uma tabela está na segunda forma normal se ela atende a todos os requisitos da primeira forma normal e se os registros na tabela, que não são chaves, dependam da chave primária em sua totalidade e não apenas parte dela. A segunda forma normal trabalha com essas irregularidades e previne que haja redundância no banco de dados. Para isso, devemos localizar os valores que dependem parcialmente da chave primária e criar tabelas separadas para conjuntos de valores que se aplicam a vários registros e relacionar estas tabelas com uma chave estrangeira. Se em algum momento tivermos que alterar o título de um filme, teríamos que procurar e alterar os valores em cada tupla (linha) da tabela. Isso demandaria um trabalho e tempo desnecessário. Porém, ao criarmos uma tabela e vincularmos elas com o recurso da chave estrangeira, tornamos o nosso banco mais organizado e ágil para as futuras consultas e manutenções que podem vir a ser necessárias.

Terceira Forma Normal: se analisarmos uma tupla e não encontrarmos um atributo não chave dependente de outro atributo não chave, podemos dizer que a entidade em questão está na terceira forma normal, contanto que esta não vá de encontro as especificações da primeira e da segunda forma normal. Como procedimento principal para configurar uma entidade que atenda as regras da terceira forma normal, nós identificamos os campos que não dependem da chave primária e sim de um outro campo não chave. Isso após separamos eles para criar uma outra tabela distinta, se necessário.

Desnormalização de dados

Desnormalização é uma técnica aplicada a bancos de dados relacionais com o objetivo de otimizar a performance de consultas que envolvem muitas tabelas. Esse tipo de consulta normalmente requer a utilização de junções (JOINS) entre tabelas para obter todos os dados necessários, o que acaba comprometendo o desempenho do banco de dados.

Para contornar esse problema em casos específicos pode ser viável desnormalizar o banco, juntando os dados em uma única tabela (ou menos tabelas do que as que eram usadas originalmente). Apesar de isso acabar gerando redundância de informações, as aplicações serão beneficiadas com o ganho de desempenho devido a não ser mais necessário unir várias tabelas.

Principais conceitos Entidade – Relacionamento

Entidades:

Entidade pode ser entendida como uma “coisa” ou algo da realidade modelada, onde se deseja manter informações no banco de dados. É uma representação concreta ou abstrata de um dos objetos, com características semelhantes do mundo real. Se algo existe e proporciona algum interesse em manter dados sobre ele, isso se caracteriza como uma Entidade do negócio. Podemos dizer que uma entidade será uma tabela em nosso banco de dados. Geralmente, quando identificamos todas as entidades, estaremos definindo quais serão as tabelas que teremos que criar em nosso banco de dados. Exemplos: Fornecedor, pessoa, imóvel, curso, aluno, professores e disciplinas.

Atributos:

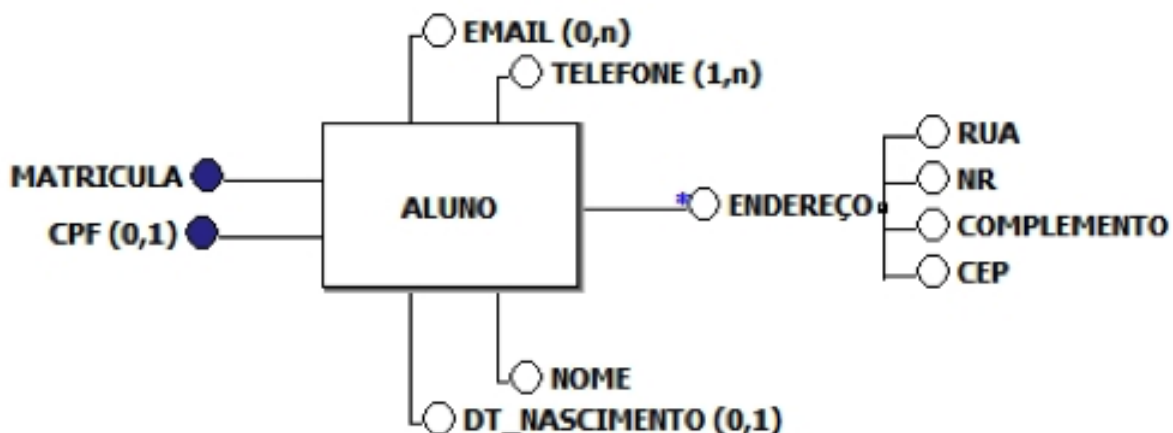
São propriedades que identificam as entidades. Os atributos podem ser simples, composto, único, não único, opcional, obrigatório, monovalorado, multivalorado ou determinante.

- **Atributo Simples:** a maioria dos atributos serão simples. Quando um atributo não é composto, recebe um valor único como nome, por exemplo: nome, sexo, data de nascimento, dentre outros.
- **Atributo Composto:** é formado por vários itens menores. Exemplo: Endereço. Seu conteúdo poderá ser dividido em vários outros atributos, como: Rua, Número, Complemento, Bairro, Cep e Cidade. Conceitualmente é aceito o endereço como um único atributo, mas na prática geralmente é feito este

desmembramento para permitir a organização dos dados inseridos e facilitar a busca e indexação dos mesmos.

- **Atributo Multivalorado:** o seu conteúdo é formado por mais de um valor, exemplo: Telefone. Uma pessoa poderá ter mais de um número de telefone.
- **Atributo Determinante:** identifica de forma única uma entidade, ou seja, não pode haver dados repetidos. Exemplo: CNPJ, CPF, Código do fornecedor, Número da matrícula etc. Devemos considerar que toda tabela no banco de dados precisa ter um atributo determinante, que também chamamos de chave primária.
- **Atributo Identificador:** identifica unicamente cada entidade de um conjunto-entidade, devem ser obrigatórios e únicos Ex.: Cod_Func.
- **Atributo Derivado:** o seu valor pode ser calculado a partir do valor de outro(s) atributo(s). Ex.: idade (pode ser calculada a partir da data de nascimento).
- **Domínio de um atributo:** descrição de possíveis valores permitidos para um atributo. Ex.: Sexo {M, F}.
- **Tipo de um Atributo:** determina a natureza dos valores permitidos para um atributo. Ex.: inteiro, real, string etc.

Figura 13 – Exemplo de Atributo.



Esquema de um Banco de Dados é a especificação da estrutura do Banco de Dados e a Instância é o conjunto de ocorrências dos objetos de dados de um esquema em um dado momento do tempo.

Relacionamentos:

Relacionamento é um conjunto de associações entre entidades.

Cardinalidade do relacionamento:

- 1:1 (um para um – uma linha de uma tabela tem apenas um relacionamento com outra linha de outra tabela. Um aluno mora atualmente em um único endereço).
- 1:N (um para n – uma linha de uma tabela pode ter “n” relacionamentos com outra tabela – um pai pode ter “n” filhos).
- N:1 (idem anterior).
- N:N ou N:M muitos para muitos (1 aluno cursa “n” disciplinas e uma disciplina pode conter “n” alunos)..

Devemos observar alguns aspectos importantes que sinalizam erros na modelagem, é importante atentar para esses erros para que não haja acúmulo de inconsistências e que não torne a modelagem um processo problemático.

1. Quando “sobram” entidades sem relacionamentos;
2. Quando ocorrem “ciclos fechados” de relacionamentos; Exemplo: Usuário relaciona-se com Empréstimo que relaciona-se com Livro que relaciona-se com Usuário que relaciona-se com Empréstimo etc.;
3. Entidades com muitos atributos relacionando-se com entidades com apenas alguns atributos;
4. Muitas entidades relacionando-se à uma mesma entidade.

Linguagem SQL

O Modelo Relacional prevê, desde sua concepção, a existência de uma linguagem baseada em caracteres que suporte a definição do esquema físico (tabelas, restrições etc.), e sua manipulação (inserção, consulta, atualização e remoção).

Os tipos da linguagem SQL são:

- DDL – Data Definition Language: Linguagem de Definição de Dados. São os comandos que interagem com os objetos do banco. São comandos DDL: CREATE, ALTER e DROP.
- DML – Data Manipulation Language - Linguagem de Manipulação de Dados. São os comandos que interagem com os dados dentro das tabelas. São comandos DML: INSERT, DELETE e UPDATE.
- DQL – Data Query Language - Linguagem de Consulta de dados. São os comandos de consulta. São comandos DQL: SELECT (é o comando de consulta). Aqui cabe um parêntese. Alguns autores consideram que o SELECT fica na DML, recentemente há a visão que o SELECT faz parte da DQL..
- DTL – Data Transaction Language - Linguagem de Transação de Dados. São os comandos para controle de transação. São comandos DTL : BEGIN TRANSACTION, COMMIT E ROLLBACK.
- DCL – Data Control Language – Linguagem de Controle de Dados. São os comandos para controlar a parte de segurança do banco de dados. São comandos DCL: GRANT, REVOKE E DENY.

Capítulo 4. Schema e Validation

A partir da versão 3.2 o MongoDB fornece a capacidade de realizar a validação do esquema durante atualizações e inserções.

- As regras de validação são por coleta.
- Para especificar as regras de validação ao criar uma nova collection, use `db.createCollection()` com a opção “validation”.
- Para adicionar validação de documento a uma collection existente, use o comando `collMod` com a opção “validation”.

O MongoDB também fornece as seguintes opções relacionadas:

- A opção `validationLevel` determina o quão estritamente o MongoDB aplica regras de validação aos documentos existentes durante uma atualização;
- A opção `validationAction` determina se o MongoDB está em estado de erro (error) e vai rejeitar documentos que violam as regras de validação ou estão em estado de warn, mas permitirá os documentos inválidos.

Database e Collections no MongoDB

O MongoDB cria data bases a partir de uma coleção de documentos (Collections), e essas collections são formadas por um agrupamento de documentos no formato JSON.

Banker, Kyle et al. (2016), explicam que o MongoDB divide as coleções em bancos de dados separados e que, ao contrário da sobrecarga habitual que os bancos de dados produzem no mundo sql, os bancos de dados no MongoDB são somente espaços de nome para distinguir entre as coleções. Para consultar no MongoDB, precisaremos saber o banco de dados (ou namespace) e a collection para qual documento se deseja consultar.

Marchioni (2015) explica que quem veio dos bancos de dados RDBMS, foi surpreendido com a possibilidade de mudar para um novo banco de dados sem tê-lo criado anteriormente. O ponto é que a criação da base de dados no MongoDB não é necessária. Os bancos de dados e as coleções são criados primeiramente quando os documentos são realmente inseridos. Sendo assim, coleções de bancos de dados individuais podem ser criadas em tempo de execução da mesma maneira que a estrutura de um documento é conhecido.

Para se criar um database basta executar o comando *use <database>*. O comando *show dbs* mostra os databases existentes. Comando *db* mostra o database corrente. Se executarmos o comando para criar uma Collection, o database também será criado.

```
db.minhaCollection.insert({"valor": 123})
```

Admin Database serve para controlar usuários e privilégios. Os usuários devem ter acesso ao banco de dados admin para executar determinados comandos administrativos.

O config Database é principalmente para uso interno e, durante as operações normais, você nunca deve inserir ou armazenar dados manualmente nele. Suporta operações de sharded cluster. Sharding é um método para distribuir dados em várias máquinas. O MongoDB usa fragmentação para dar suporte a implantações com conjuntos de dados muito grandes e operações de alto rendimento.

Em cada instância do mongod o Local Database tem seu próprio banco de dados local, que armazena os dados usados no processo de replicação e outros dados específicos da instância *local.startup_log*.

Para se criar uma coleção, basta inserir um documento nela ou executar o comando *db.createCollections()*.

```
Exemplo: db.customers.insert({ nome: "sue", idade: 26, status: "A" })
```

Se quiser verificar quais coleções existem no banco de dados é só executar o comando *show collections* ou *show tables*.

JSON Schema

A partir da versão 3.6, o MongoDB suporta a validação de JSON Schema. Para especificar a validação do JSON Schema, use o operador de `$jsonSchema` na sua expressão “`validator`”.

O exemplo a seguir especifica regras de validação usando o esquema JSON.

```
db.createCollection("students", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: [ "name", "year", "major", "address" ],
      properties: {
        name: {
          bsonType: "string",
          description: "must be a string and is required"
        },
        year: {
          bsonType: "int",
          minimum: 2017,
          maximum: 3017,
          description: "must be an integer in [ 2017, 3017 ] and is
required"
        },
        major: {
          enum: [ "Math", "English", "Computer Science", "History",
null ],
          description: "can only be one of the enum values and is r
equired"
        },
        gpa: {
```



```

    bsonType: [ "double" ],
    description: "must be a double if the field exists"
  },
  address: {
    bsonType: "object",
    required: [ "city" ],
    properties: {
      street: {
        bsonType: "string",
        description: "must be a string if the field exists"
      },
      city: {
        bsonType: "string",
        description: "must be a string and is required"
      }
    }
  }
}
}
})

```

Além da validação do JSON Schema que utiliza o operador de consulta `$jsonSchema`, o MongoDB suporta validação com outros operadores de consulta, com exceção de:

- `$near`,
- `$nearSphere`,
- `$text`,
- `$where`, e
- `$expr` com a expressão `$function`.

O exemplo a seguir especifica regras de validação usando a expressão de consulta:

```
db.createCollection( "contacts",
  { validator: { $or:
    [
```

```
{ phone: { $type: "string" } },
{ email: { $regex: /@mongodb\.com$/ } },
{ status: { $in: [ "Unknown", "Incomplete" ] } }
]
}
} )
```

A validação ocorre durante atualizações e inserções. Quando você adiciona validação a uma coleção, os documentos existentes não passam por verificações de validação até a modificação.

Em documentos existentes a opção *validationLevel* determina em quais operações o MongoDB aplica as regras de validação:

- Se *validationLevel* = “strict” (o padrão), o MongoDB aplica regras de validação a todas as inserções e atualizações.
- Se *validationLevel* = “moderate”, o MongoDB aplica regras de validação para inserções e atualizações de documentos existentes que já cumprem os critérios de validação. Com o nível moderado, atualizações aos documentos existentes que não preenchem os critérios de validação não são verificadas.

Por exemplo, crie uma collection *contacts* com os seguintes documentos:

```
db.contacts.insert([
  { "_id": 1, "name": "Anne", "phone": "+1 555 123 456", "city":
"London", "status": "Complete" },
  { "_id": 2, "name": "Ivan", "city": "Vancouver" }
])
```

Escreva o seguinte comando para adicionar um validador a collection *contacts*:

```
db.runCommand( {
  collMod: "contacts",
  validator: { $jsonSchema: {
    bsonType: "object",
```

```

    required: [ "phone", "name" ],
    properties: {
      phone: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      name: {
        bsonType: "string",
        description: "must be a string and is required"
      }
    }
  } },
  validationLevel: "moderate"
} )

```

A collection `contacts` agora tem um validador com *validationLevel moderate*.

Se você tentar atualizar o documento com `_id 1`, o MongoDB aplicaria as regras de validação, uma vez que o documento existente corresponde aos critérios. Em contrapartida, o MongoDB não aplicará regras de validação às atualizações do documento com `_id 2`, pois não cumpre as regras de validação.

Para desativar totalmente a validação, você pode *definir validationLevel off*.

Para aceitar ou rejeitar documentos inválidos temos a opção *validationAction* que determina como o MongoDB lida com documentos que violam as regras de validação:

Se *validationAction* é *error* (o padrão), o MongoDB rejeita qualquer inserção ou atualização que viole os critérios de validação.

Se *validationAction* é *warn*, o MongoDB registra quaisquer violações, mas permite que a inserção ou atualização prossiga.

Por exemplo, crie uma collection `contacts2` com o seguinte validador JSON Schema:

```

db.createCollection( "contacts2", {
  validator: { $jsonSchema: {
    bsonType: "object",
    required: [ "phone" ],
    properties: {
      phone: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      email: {
        bsonType : "string",
        pattern : "@mongodb\\.com$",
        description: "must be a string and match the regular expres
sion pattern"
      },
      status: {
        enum: [ "Unknown", "Incomplete" ],
        description: "can only be one of the enum values"
      }
    }
  } },
  validationAction: "warn"
} )

```

Com o *validationAction warn*, o MongoDB registra quaisquer violações, mas permite que a inserção ou atualização prossiga.

Por exemplo, a seguinte operação de inserção viola a regra de validação:

```

db.contacts2.insert( { name: "Amanda", status: "Updated" } )

```

No entanto, se *validationAction* é *warn*, o MongoDB registra apenas a mensagem de violação de validação e permite que a operação prossiga:

```

2017-12-01T12:31:23.738-05:00 W STORAGE [conn1] Document would fail validation collection: example.contacts2 doc: { _id: ObjectId('5a2191ebacbbfc2bdc4dcffc'), name: "Amanda", status: "Updated" }

```

Collection com Validação

Uma collection pode ser criada observando parâmetros informados. Você pode criar uma collection do tipo “object” onde podem ser informados os fields obrigatórios e as propriedades para cada field, conforme abaixo:

```
db.createCollection("students", {  
  validator: {  
    $jsonSchema: {  
      bsonType: "object",  
      required: [ "name", "year", "major", "address" ],  
      properties: {  
        name: {  
          bsonType: "string",  
          description: "must be a string and is required"  
        },  
        year: {  
          bsonType: "int",  
          minimum: 2017,  
          maximum: 3017,  
          description: "must be an integer in [ 2017, 3017 ] and is required"  
        }  
      }  
    }  
  }  
}
```

}}

Como exemplo, podemos rodar o comando da imagem abaixo para criar uma collection “carro” e suas validações.

```
MongoDB Enterprise > db.carro.drop()
true
MongoDB Enterprise > db.createCollection("carro", {
...   validator: {
...     $jsonSchema: {
...       bsonType: "object",
...       required: ["modelo", "ano"],
...       properties: {
...         modelo: {
...           bsonType: "string",
...           description: "Modelo é string e é obrigatório."
...         },
...         fabricante: {
...           bsonType: "string",
...           minLength: 3,
...           maxLength: 30,
...           description: "Deve ser string"
...         },
...         ano: {
...           bsonType: "int",
...           minimum: 2015,
...           maximum: 2025,
...           description: "Inteiro entre [ 2015, 2025 ] e é obrigatório."
...         }
...       }
...     }
...   }
... })
{ "ok" : 1 }
```

Ao tentar inserir um documento na collection carro, teremos um erro quanto ao ano 2020 porque o Mongo interpreta que esse valor é do tipo double e o field é do tipo int.

```
MongoDB Enterprise > db.carro.insert({modelo: 'HB20', fabricante: 'Hyundai', ano: 2020})
WriteResult({
  "nInserted" : 0,
  "writeError" : {
    "code" : 121,
    "errmsg" : "Document failed validation"
  }
})
```

Para resolver isso, basta utilizar a função “*NumberInt*” no valor do ano informado.

```
MongoDB Enterprise > db.carro.insert({modelo: 'HB20', fabricante: 'Hyundai', ano: NumberInt(2020)})
WriteResult({ "nInserted" : 1 })
```

Repare que se tentarmos inserir um documento onde o fabricante tem apenas 2 caracteres teremos um erro, pois há uma validação no qual o fabricante tem que ter entre 3 e 30 caracteres.

```
MongoDB Enterprise > db.carro.insert({modelo: 'HB20', fabricante: 'Hyundai', ano: NumberInt(2020)})
WriteResult({ "nInserted" : 1 })
MongoDB Enterprise > db.carro.insert({modelo: 'KA', fabricante: 'Ford', ano: NumberInt(2017)})
WriteResult({ "nInserted" : 1 })
MongoDB Enterprise > db.carro.insert({modelo: 'S10', fabricante: 'GM', ano: NumberInt(2017)})
WriteResult({
  "nInserted" : 0,
  "writeError" : {
    "code" : 121,
    "errmsg" : "Document failed validation"
  }
})
```

Tentar inserir o ano como string também gera erro.

```
MongoDB Enterprise > db.carro.insert({modelo: 'Onix', fabricante: 'Chevrolet', ano: '2017'})
WriteResult({
  "nInserted" : 0,
  "writeError" : {
    "code" : 121,
    "errmsg" : "Document failed validation"
  }
})
MongoDB Enterprise > db.carro.insert({modelo: 'Onix', fabricante: 'Chevrolet', ano: NumberInt('2017')})
WriteResult({ "nInserted" : 1 })
```

Capped Collection

Capped collections são collections circulares de tamanho fixo que seguem a ordem de inserção para suportar alto desempenho para criar, ler e excluir operações. Por circular, entende-se que quando o tamanho fixo alocado para a coleção estiver esgotado, ele começará a excluir o documento mais antigo da coleção sem fornecer quaisquer comandos explícitos. Capped collections restringem atualizações aos documentos se a atualização resultar em maior tamanho do documento. Uma vez que as coleções limitadas armazenam documentos na ordem do armazenamento do disco, ele garante que o tamanho do documento não aumente o tamanho alocado no disco. Coleções limitadas são as melhores para armazenar informações de registro, dados de cache ou quaisquer outros dados de alto volume.

```
db.createCollection( <name>,
{
  capped: <boolean>,
  autoIndexId: <boolean>,
  size: <number>,
  max: <number>,
  storageEngine: <document>,
  validator: <document>,
  validationLevel: <string>,
  validationAction: <string>,
  indexOptionDefaults: <document>,
  viewOn: <string>, // Added in MongoDB 3.4
  pipeline: <pipeline>, // Added in MongoDB 3.4
  collation: <document>, // Added in MongoDB 3.4
  writeConcern: <document>
}
)
```

Capped: boolean, opcional. Para criar uma coleção limitada, especifique *true*. Se você especificar *true*, também deve definir um tamanho máximo no campo *size*.

Size: *number* opcional. Especifique um tamanho máximo em bytes para uma coleção limitada. Quando uma coleção limitada atinge seu tamanho máximo, o MongoDB remove os documentos mais antigos para liberar espaço para os novos documentos. O campo de tamanho é obrigatório para coleções limitadas e ignorado para outras coleções.

Max: *number* opcional. O número máximo de documentos permitidos na coleção limitada. O limite de tamanho tem precedência sobre esse limite. Se uma coleção limitada atingir o limite de tamanho antes de atingir o número máximo de documentos, o MongoDB remove os documentos antigos. Se você preferir usar o limite máximo, certifique-se de que o limite de tamanho, que é necessário para uma coleção limitada, seja suficiente para conter o número máximo de documentos.

Como dito, o limite de tamanho tem precedência sobre esse limite. Observe o exemplo abaixo onde vamos limitar pelo **Max** = 5 e faremos insert de 6 documentos para observar o comportamento do Capped Collection.

```
db.createCollection("log_simulado", {
```



```

        capped: true,

        size: 1024, // em bytes

        max: 5

    })

    db.log_simulado.insert({n: 1, desc: '1xxxxxx'})
    db.log_simulado.insert({n: 2, desc: '2xxxxxx'})
    db.log_simulado.insert({n: 3, desc: '3xxxxxx'})
    db.log_simulado.insert({n: 4, desc: '4xxxxxx'})
    db.log_simulado.insert({n: 5, desc: '5xxxxxx'})
    db.log_simulado.insert({n: 6, desc: '6xxxxxx'})

```

```

MongoDB Enterprise > db.createCollection("log_simulado", {
... capped: true,
... size: 1024, // em bytes
... max: 5
... })
{ "ok" : 1 }
MongoDB Enterprise > db.log_simulado.insert({n: 1, desc: '1xxxxxx'})
WriteResult({ "nInserted" : 1 })
MongoDB Enterprise > db.log_simulado.insert({n: 2, desc: '2xxxxxx'})
WriteResult({ "nInserted" : 1 })
MongoDB Enterprise > db.log_simulado.insert({n: 3, desc: '3xxxxxx'})
WriteResult({ "nInserted" : 1 })
MongoDB Enterprise > db.log_simulado.insert({n: 4, desc: '4xxxxxx'})
WriteResult({ "nInserted" : 1 })
MongoDB Enterprise > db.log_simulado.insert({n: 5, desc: '5xxxxxx'})
WriteResult({ "nInserted" : 1 })
MongoDB Enterprise > db.log_simulado.find()
{ "_id" : ObjectId("5ff08879c6128ce3f046816b"), "n" : 1, "desc" : "1xxxxxx" }
{ "_id" : ObjectId("5ff08880c6128ce3f046816c"), "n" : 2, "desc" : "2xxxxxx" }
{ "_id" : ObjectId("5ff08886c6128ce3f046816d"), "n" : 3, "desc" : "3xxxxxx" }
{ "_id" : ObjectId("5ff0888bc6128ce3f046816e"), "n" : 4, "desc" : "4xxxxxx" }
{ "_id" : ObjectId("5ff08890c6128ce3f046816f"), "n" : 5, "desc" : "5xxxxxx" }
MongoDB Enterprise > db.log_simulado.insert({n: 6, desc: '6xxxxxx'})
WriteResult({ "nInserted" : 1 })
MongoDB Enterprise > db.log_simulado.find()
{ "_id" : ObjectId("5ff08880c6128ce3f046816c"), "n" : 2, "desc" : "2xxxxxx" }
{ "_id" : ObjectId("5ff08886c6128ce3f046816d"), "n" : 3, "desc" : "3xxxxxx" }
{ "_id" : ObjectId("5ff0888bc6128ce3f046816e"), "n" : 4, "desc" : "4xxxxxx" }
{ "_id" : ObjectId("5ff08890c6128ce3f046816f"), "n" : 5, "desc" : "5xxxxxx" }
{ "_id" : ObjectId("5ff088a1c6128ce3f0468170"), "n" : 6, "desc" : "6xxxxxx" }

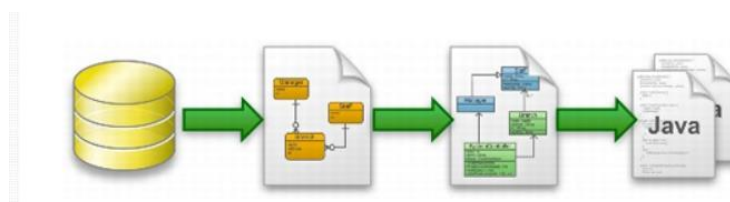
```

ORM, ODM

Qual o melhor jeito de interagir (conectar) com um banco de dados? Existem duas abordagens:

- Usando a linguagem de consulta nativa dos bancos de dados (por exemplo, SQL).
- Usando um ORM ou um ODM.

Segundo Krakowiak (2003), em um sistema de computação distribuída, middleware é definido como uma camada de software que reside entre o sistema operacional e as aplicações em cada ponto do sistema.



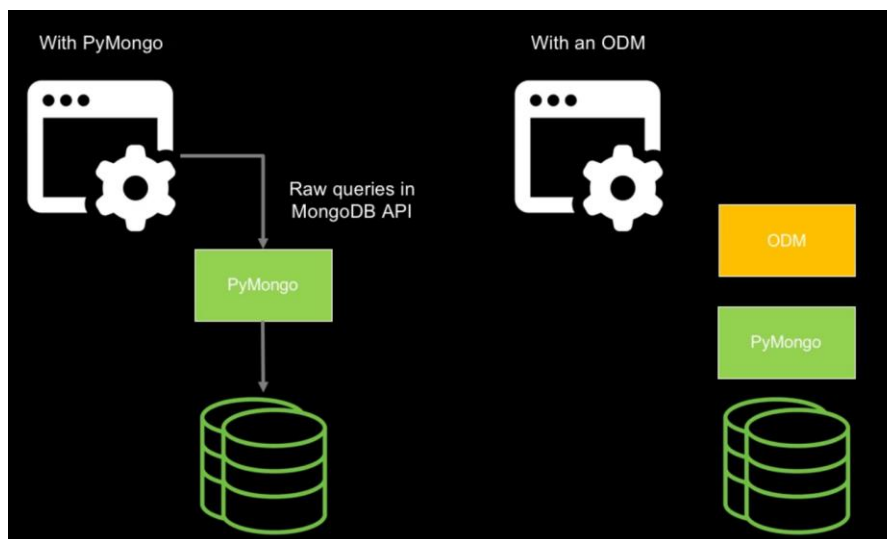
- ORM (Mapeamento Objeto Relacional).
- ODM (Mapeador Documento Objeto).

São técnicas de programação para conversão de dados entre sistemas de tipos incompatíveis em bancos de dados e linguagens de programação orientadas a objetos. Isso cria um "banco de dados de objeto virtual" que pode ser usado a partir de dentro da linguagem de programação.

- ORM - para bancos de dados relacionais.
- ODM - para NoSQL bancos de dados.

ORM (Object Relational Mapper) significa mapear um objeto com um mundo relacional. Basicamente converte dados entre tipos incompatíveis em linguagens de programação orientadas a objetos. ORM envolve os detalhes específicos de implementação de drivers de armazenamento em uma API (interface de programa de aplicativo) e mapeia os campos relacionais para membros de um objeto.

ODM (Object Document Mapper) é um “mapeador” de documentos do MongoDB.



Um melhor desempenho pode ser obtido usando SQL ou qualquer linguagem de consulta suportada pelo banco de dados.

ORM / ODMs são frequentemente mais lentos porque usam código de tradução para mapear entre objetos e o formato de banco de dados, o que pode não usar as consultas de banco de dados mais eficientes.

O benefício de usar um ORM / ODM é que os programadores podem continuar a pensar em termos de objetos JavaScript em vez da semântica do banco de dados – isto é particularmente verdadeiro se você precisar trabalhar com bancos de dados diferentes (no mesmo site ou em diferentes sites).

Eles também fornecem um local centralizado para realizar a validação e verificação de dados.

Existem muitas soluções ODM / ORM disponíveis:

- **Mongoose:** ferramenta de modelagem de objetos MongoDB projetada para funcionar em um ambiente assíncrono.

- **Waterline:** é um ORM extraído da estrutura *web Sails.js* baseada no Express. Ele fornece uma API uniforme para acessar vários bancos de dados diferentes, incluindo Redis, MySQL, LDAP, MongoDB e Postgres.
- **Bookshelf.js:** apresenta interfaces de retorno de chamada tradicionais e baseadas em promessa, fornecendo suporte a transações, carregamento de relação ansioso/aninhado, associações polimórficas e suporte para relações um-para-um, um-para-muitos e muitos-para-muitos. Funciona com PostgreSQL, MySQL e SQLite3.
- **Objection.js:** torna o mais fácil possível o uso de todo o poder do SQL e do mecanismo de banco de dados subjacente (suporta SQLite3, Postgres e MySQL).
- **Sequelize:** ORM para Node.js e io.js. Ele suporta PostgreSQL, MySQL, MariaDB, SQLite e MSSQL e oferece suporte a transações sólidas, relações, replicação de leitura entre outros.
- **Node ORM2:** é um ORM para NodeJS. Ele suporta MySQL, SQLite e Progress, ajudando a trabalhar com o banco de dados usando uma abordagem orientada a objetos.
- **JugglingDB:** é um ORM de banco de dados cruzado para NodeJS, fornecendo uma interface comum para acessar os formatos de banco de dados mais populares. Atualmente com suporte a MySQL, SQLite3, Postgres, MongoDB, Redis e js-memory-storage (mecanismo de escrita própria apenas para teste).

O Mongoose é o mais popular ODM no momento e é uma escolha razoável se você estiver usando o MongoDB para seu banco de dados. O Mongoose fornece um método simples e baseado em solução de esquema para modelar seus dados no aplicativo. Inclui seleção de tipo, validação, construção de consulta e lógica de negócios. O Mongoose é instalado no seu projeto (package.json) assim como outra dependência qualquer — usando NPM. Para instalá-lo, use a seguinte linha de comando dentro da pasta do seu projeto: `npm install mongoose`. A instalação do

Mongoose adiciona todas as suas dependências, incluindo o driver de banco de dados MongoDB, mas não instala o MongoDB propriamente dito.

Além de definir a estrutura de seus documentos e os tipos de dados que você está armazenando, um Schema lida com a definição de:

- Validators (async and sync);
- Defaults;
- Getters;
- Setters;
- Indexes;
- Middleware;
- Methods definition;
- Statics definition;
- Plugins;
- Pseudo-JOINs.

Abaixo um exemplo de esquema no Mongoose.

```
const Comment = new Schema({
  name: { type: String, default: 'hahaha' },
  age: { type: Number, min: 18, index: true },
  bio: { type: String, match: /[a-z]/ },
  date: { type: Date, default: Date.now },
  buff: Buffer
});

// a setter
Comment.path('name').set(function (v) {
  return capitalize(v);
});

// middleware
Comment.pre('save', function (next) {
  notify(this.get('email'));
  next();
});
```

Capítulo 5. Armazenamento de Dados em Nuvem e Sistemas de Arquivos

Introdução ao Banco de Dados em Nuvem - DBaaS

O uso de Cloud Computing pelas empresas já se popularizou graças a seus inúmeros benefícios. O Banco de Dados Híbridos, ou Database as a Service (DBaaS), assim como os outros sistemas que funcionam como serviço, pode ajudar uma organização em seus processos. O DBaaS é baseado em sistemas na Nuvem e é capaz de oferecer plataformas flexíveis, escalonadas e sob demanda para seus usuários. A entrega do software de banco de dados é feita pelo fornecedor do serviço, que é responsável por alocar fisicamente o Data Center. Esse tipo de serviço tem se tornado, cada vez mais essencial para os negócios, visto que a maioria das empresas já migraram para a Cloud, aumentando a importância de se ter uma base de dados online.

Os bancos de dados baseados na nuvem permitem que os usuários armazenem, gerenciem e recuperem seus dados estruturados, não estruturados e semiestruturados, por meio de uma plataforma na nuvem acessível pela Internet. Os bancos de dados em nuvem são também conhecidos como banco de dados como serviço (DBaaS), pois normalmente são oferecidos como serviços gerenciados.

Por que usar bancos de dados na nuvem?

- **Os custos de TI podem ser reduzidos.** A empresa não precisa investir, manter e dar suporte a um ou mais bancos de dados.
- **Tecnologias automatizadas.** Seu banco de dados colhe benefícios de diversos processos automatizados, como fallover automático, recuperação de falha e auto-dimensionamento.
- **Maior acessibilidade.** Desde que haja conexão com a Internet, é possível acessar seu banco de dados na nuvem a partir de qualquer lugar, a qualquer momento.
- **Experiência em TI.** Os projetos de banco de dados são famosos por serem desafiadores. Você pode aprimorar as habilidades relacionadas a banco de

dados da sua equipe de TI com as de um provedor de serviços, ou confiar totalmente na experiência em banco de dados da equipe do provedor de serviços.

- As principais características do DBaaS são:
- **Disponibilidade sob demanda:** o serviço deve estar disponível 24 horas por dia, 7 dias por semana para os usuários. Não há a necessidade de se instalar e configurar hardwares ou softwares. A utilização é totalmente virtual.
- **Pagamento por assinatura:** a empresa que contratar o DBaaS realizará o pagamento por meio de uma assinatura, conforme estabelecido em contrato. Normalmente, o pagamento é mensal e de acordo com a capacidade de armazenamento utilizada, processamento e desempenho.
- **O Fornecedor é responsável pela gestão:** uma das principais características desse serviço na Nuvem é que a responsabilidade de todo o gerenciamento é do fornecedor. Assim, a empresa não precisa se preocupar em manter, atualizar ou administrar seu banco de dados.

O grande diferencial do DBaaS em relação a um banco de dados comum é seu autoatendimento, que torna a gestão e uso das ferramentas uma tarefa fácil. Ele ainda permite que os usuários consumam, operem e configurem o banco de dados por meio de conjuntos de abstrações, mesmo que não tenham conhecimento técnico para isso.

Movimentando seus bancos de dados para a Nuvem

Como a maioria das transformações organizacionais, mover para a nuvem não é algo que se realiza da noite para o dia. Você deve escolher um projeto para experimentar em um provedor de nuvem escolhido. Quase todas as organizações que se movem para a nuvem fazem uma POC (prova de conceito) com um banco de dados não crítico.

Embora cada migração seja única, você provavelmente seguirá as etapas:

Planejamento:

- Coleta de requisitos.
- Determinação de recursos para atender aos requisitos.
- Avaliação de quais bancos de dados mover e quais mudanças podem ser necessárias para o banco de dados ou os aplicativos que o utilizam.
- Estabelecer critérios de sucesso e critérios de reversão.

Movimentando os dados:

- Replicação.
- Incorporação de mudanças desde que a réplica foi criada.
- Teste de aplicação.
- Verificações pós-migração.

Otimização:

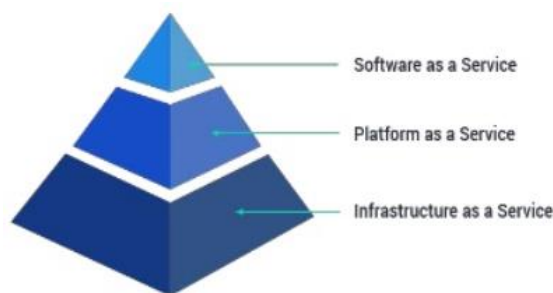
- Ajuste de desempenho.
- Projetando alta disponibilidade.
- Determinar quais eventos registrar e monitorar.
- Criação de um plano de recuperação de desastres.

Modelos em Nuvem

Para se falar de Nuvem é importante abordar os três tipos de modelos em nuvem:

- SaaS é caracterizado principalmente pela não-aquisição de licenças de software.
- PaaS envolve um ambiente virtual para criação, hospedagem e controle de softwares e bancos de dados.
- IaaS apenas abstrai aspectos relacionados à parte física de servidores e redes.

Figura 14 – Modelos de Nuvem.



IaaS — Infrastructure as a Service (Infraestrutura como Serviço):

Nesse exemplo dos modelos de nuvem, a empresa contrata uma capacidade de hardware que corresponde a memória, armazenamento, processamento, etc. Podem entrar nesse pacote de contratações os servidores, roteadores, racks, entre outros.

Dependendo do fornecedor e do modelo escolhido, a sua empresa pode ser tarifada, por exemplo, pelo número de servidores utilizados e pela quantidade de dados armazenados ou trafegados. Em geral, tudo é fornecido por meio de um data center com servidores virtuais, em que você paga somente por aquilo que usar.

PaaS — Platform as a Service (Plataforma como Serviço):

Nesse cenário, o PaaS surge como o ideal porque é, como o próprio nome diz, uma plataforma que pode criar, hospedar e gerir esse aplicativo. Nesse modelo de nuvem, contrata-se um ambiente completo de desenvolvimento, no qual é possível criar, modificar e otimizar softwares e aplicações.

Aqui, a grande vantagem é que a equipe de desenvolvimento só precisa se preocupar com a programação do software, pois o gerenciamento, manutenção e atualização da infraestrutura ficam a cargo do fornecedor e as várias ferramentas de desenvolvimento de software são oferecidas na plataforma.

SaaS — Software as a Service (Software como Serviço):

Nesse modelo de nuvem você pode ter acesso a um software sem comprar a sua licença, utilizando-o a partir da Cloud Computing.

Muitos CRMs ou ERPs trabalham no sistema SaaS. Assim, o acesso a esses softwares é feito usando a internet. Os dados, contatos e demais informações podem ser acessados de qualquer dispositivo, dando mais mobilidade à equipe.

O Google Docs e o Office 365 funcionam dessa maneira.

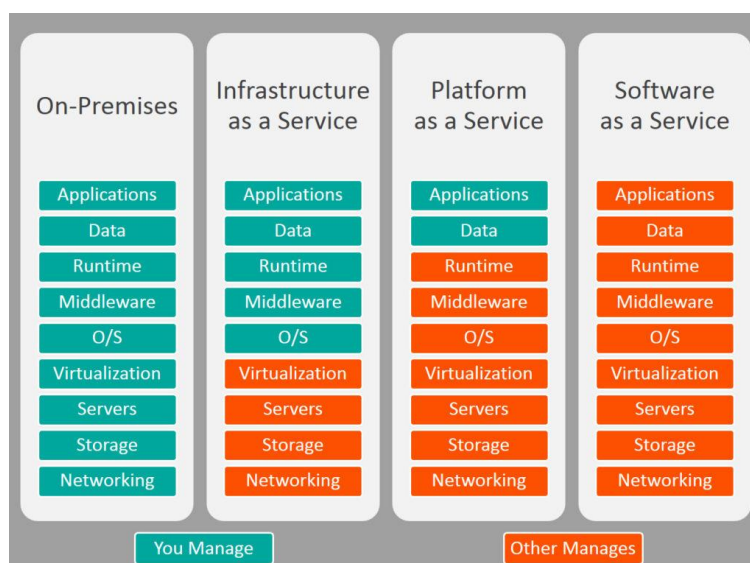
On-Premises:

Um servidor on-premises é aquele em que a própria empresa tem a responsabilidade de processar suas aplicações de hardware e software.

Em outras palavras, toda a infraestrutura, customização, configuração e atualização é feita internamente.

A figura abaixo denota as responsabilidades de gerenciamento para cada modelo em Nuvem.

Figura 15 – Gerenciamento modelo em Nuvem.
















Introdução a AWS

A Amazon Web Services, também conhecida como AWS, é uma plataforma de serviços de computação em nuvem que formam uma plataforma de computação na nuvem oferecida pela Amazon.com. Os serviços são oferecidos em várias geográficas distribuídas pelo mundo.

EC2: o Amazon Elastic Compute Cloud (Amazon EC2) oferece uma capacidade de computação escalável na Nuvem da Amazon Web Services (AWS). O uso do Amazon EC2 elimina a necessidade de investir em hardware inicialmente, portanto, você pode desenvolver e implantar aplicativos com mais rapidez. Você pode usar o Amazon EC2 para executar o número de servidores virtuais que precisar, configurar a segurança e a rede, e gerenciar o armazenamento. O Amazon EC2 também permite a expansão ou a redução para gerenciar as alterações de requisitos ou picos de popularidade, reduzindo, assim, a sua necessidade de prever o tráfego do servidor.

Abaixo temos uma visão geral de instâncias indicadas por situação.

General Purpose	Compute Optimised	Memory Optimised	Accelerated Computing	Storage Optimised
 A1 ARM based core and custom silicon	 C4 Compute - CPU intensive apps and DBs	 R4 RAM - Memory intensive apps and DB's	 P2 Processing optimised - Machine Learning	 H1 High Disk Throughput - Big data clusters
 T2 Tiny - Web servers and small DBs		 X1 Xtreme RAM - For SAP/Spark	 G3 Graphics Intensive - Video and streaming	 I3 IOPS - NoSQL DBs
 M4 Main - App servers and general purpose		 z1d High Compute and High Memory - Gaming	 F1 Field Programmable - Hardware acceleration	 D2 Dense Storage - Data Warehousing

AWS – S3:

O Amazon Simple Storage Service é armazenamento para a Internet. Ele foi projetado para facilitar a computação de escala na web para os desenvolvedores. O Amazon S3 tem uma interface simples de serviços da web que você pode usar para armazenar e recuperar qualquer quantidade de dados, a qualquer momento, em qualquer lugar da web.

Ela concede acesso a todos os desenvolvedores para a mesma infraestrutura altamente dimensionável, confiável, segura, rápida e econômica que a Amazon utiliza para rodar a sua própria rede global de sites da web. O serviço visa maximizar os benefícios de escala e poder passar esses benefícios para os desenvolvedores.

AWS – RDS:

O Amazon Relational Database Service (Amazon RDS) facilita a configuração, a operação e a escalabilidade de bancos de dados relacionais na nuvem. O serviço oferece capacidade econômica e redimensionável e automatiza tarefas demoradas de administração, como provisionamento de hardware, configuração de bancos de dados, aplicação de patches e backups. Dessa forma, você pode se concentrar na performance rápida, alta disponibilidade, segurança e conformidade que os aplicativos precisam.

Mecanismos de banco de dados do Amazon RDS



As formas de cobrança da AWS podem ser:

- On-demand: O cálculo é feito em cima de horas ou segundos utilizados (no mínimo 60 segundos) e somente as instâncias EC2 que forem utilizadas.
- Instâncias reservadas (RIs): O preço por hora é fixo, independentemente do uso, e existe um prazo pré-determinado de contratação. Essa forma de pagamento tem o benefício de obter desconto, uma vez que o cliente tem o compromisso de um a três anos.
- Instâncias spot: Por serem instâncias extras, ou seja, instâncias de capacidade extra na Nuvem AWS, o preço é muito mais atrativo. Por outro lado, se o EC2 precisar de capacidade, o cliente que utiliza Instância Spot será notificado 2 minutos antes de suas instâncias serem interrompidas.

Google Cloud Platform

Google Cloud Platform é uma suíte de computação em nuvem oferecida pelo Google, funcionando na mesma infraestrutura que a empresa usa para seus produtos dirigidos aos usuários, dentre eles o Buscador Google e o YouTube.

Juntamente com um conjunto de ferramentas de gerenciamento modulares, fornecem uma série de serviços incluindo, computação, armazenamento de dados, análise de dados e aprendizagem de máquina.

Compute Engine

O Google Compute Engine é o componente Infraestrutura como serviço do Google Cloud Platform, construído sobre a infraestrutura global que executa o mecanismo de pesquisa do Google, Gmail, YouTube e outros serviços.

O Google Compute Engine permite que os usuários iniciem máquinas virtuais sob demanda. Uma máquina virtual é um computador emulado em outra máquina. Nesse caso, o servidor em nuvem executa vários sistemas operacionais em diversas outras máquinas, em vez de cada computador ter um sistema operacional próprio.

O Google Compute Engine oferece máquinas virtuais em execução nos centros de dados inovadores do Google e na rede mundial de fibra. O suporte a ferramentas e fluxo de trabalho do Compute Engine permite dimensionar de instâncias únicas para computação em nuvem balanceada de carga global. As VMs do Compute Engine são inicializadas rapidamente, vêm com opções de disco persistentes e locais de alto desempenho, além de oferecerem desempenho consistente.

- Instâncias.
- Configurações.
- Deploy de aplicação e Jobs.
- Alertas automáticos.

Storage & Databases:

A ideia de Plataforma como um Serviço (PaaS) já é bem difundida em muitos setores, especialmente no que diz respeito à gestão e ao armazenamento de dados. É similar ao Google Drive, porém, em escala bem maior, o que permite a você armazenar e organizar todos os arquivos da empresa e acessá-los a partir de qualquer máquina com permissão de acesso. É ótimo para evitar a perda de documentos e para minimizar o uso do espaço físico em seu negócio.

Armazenamento de produtos escaláveis, resilientes e de alto desempenho em Bancos de dados para suas aplicações.

- Armazenamento de arquivos.
- Gerenciamento de buckets.

- Bucket Locations.
- Multi Regional.
- Regional.
- NearLine.
- CodeLine.
- Linha de comando, API, Console.

App Engine:

Plataforma para criação de aplicativos Web escaláveis e backends para dispositivos móveis. O App Engine fornece serviços integrados e APIs, como datastores NoSQL, memcache e uma API de autenticação de usuário, comum à maioria dos aplicativos.

- Construir aplicações de desenvolvimentos escaláveis.
- Instâncias automáticas.
- Integrações.
- Deploy automatizado.

Big Data:

Quando precisamos tomar uma decisão importante para a empresa, é importante ter dados que deem suporte a essa escolha. Ainda mais, atualmente, em um mercado cada vez mais complexo e volátil.

Totalmente gerenciado, data warehousing, lote e processamento de fluxo, exploração de dados, Hadoop / Spark, e mensagens confiáveis.

- Análise de dados em bancos NoSQL.
- Construir bancos via: linha de comando, console e API's.

- Importar dados: csv, txt, integrações.

Machine Learning:

Serviços de ML rápidos, escaláveis e fáceis de usar. Use modelos pré-treinados ou treine modelos personalizados em seus dados.

- Máquina de conhecimento, como funciona.
- Exemplo Case Google: E-mail e Spam.
- Google Cloud Vision API.
- Google Translate API.
- Google Speech API.

Microsoft Azure

O Microsoft Azure é uma plataforma destinada à execução de aplicativos e serviços, baseada nos conceitos da computação em nuvem.

A apresentação do serviço foi feita no dia 27 de outubro de 2008 durante a Professional Developers Conference, em Los Angeles e ele foi lançado em 1 de fevereiro de 2010 como Windows Azure, para então ser renomeado como Microsoft Azure em 25 de março de 2014.

Sua computação em nuvem é definida como uma combinação de software como serviço (SaaS) com computação em grid. A computação em grid dá o poder de computação e alta escalabilidade oferecida para as aplicações, através de milhares de máquinas (hardware) disponíveis em centros de processamento de dados de última geração. De software como serviço se tem a capacidade de contratar um serviço e pagar somente pelo uso, permitindo a redução de custos operacionais, com uma configuração de infraestrutura realmente mais aderente às necessidades.

Além dos recursos de computação, armazenamento e administração oferecidos pelo Microsoft Azure, a plataforma também disponibiliza uma série de serviços para a construção de aplicações distribuídas, além da total integração com a solução on-premise (local) baseada em plataforma .NET.

Entre os principais serviços da plataforma Windows Azure há o SQL Azure Database, Azure AppFabric Platform e uma API de gerenciamento e monitoramento para aplicações colocadas na nuvem.

As formas de cobrança da Azure podem ser:

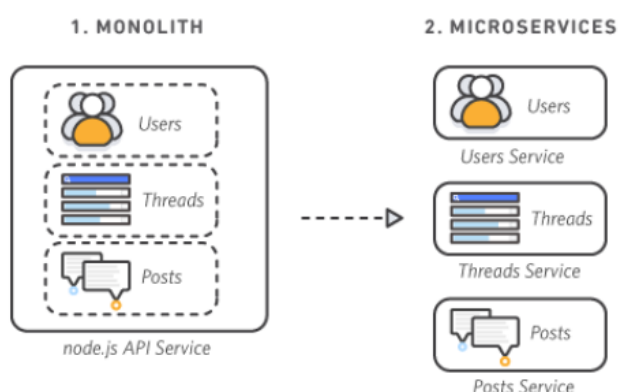
- On-demand: seus custos são realizados em cima dos minutos utilizados. Nesse modelo, não é necessário compromisso de tempo mínimo de contratação. Como o pagamento é feito de acordo com a utilização, é possível aumentar e diminuir recursos sem limite.
- Contrato pré-definido: como um determinado tempo de utilização é acordado, o custo é reduzido.
- Acordo empresarial: nessa modalidade, o pagamento é realizado antecipadamente, por esse motivo, há benefícios e desconto. O uso adicional é pago de forma separada, mas com desconto nas taxas.

Arquitetura Microsserviços

Arquiteturas Monolíticas: com as arquiteturas monolíticas, todos os processos são altamente acoplados e executam como um único serviço. Isso significa que se um processo do aplicativo apresentar um pico de demanda, toda a arquitetura deverá ser escalada. A complexidade da adição ou do aprimoramento de recursos de aplicativos monolíticos aumenta com o crescimento da base de código. Essa complexidade limita a experimentação e dificulta a implementação de novas ideias. As arquiteturas monolíticas aumentam o risco de disponibilidade de aplicativos, pois muitos processos dependentes e altamente acoplados aumentam o impacto da falha de um único processo.

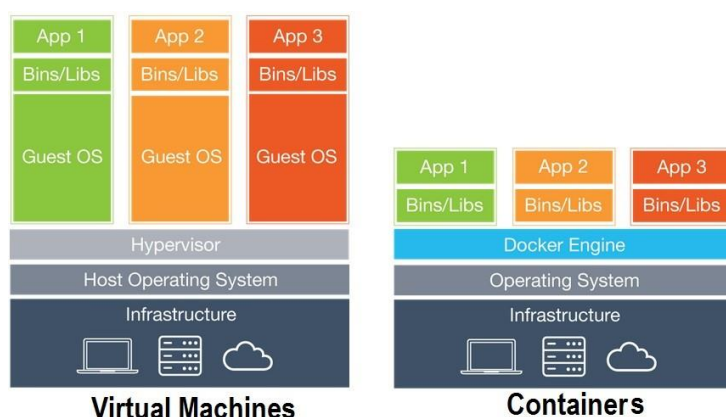
Arquitetura de Microserviços: com uma arquitetura de microserviços, um aplicativo é criado como componentes independentes que executam cada processo do aplicativo como um serviço. Esses serviços se comunicam por meio de uma interface bem definida usando APIs leves. Os serviços são criados para recursos empresariais e cada serviço realiza uma única função. Como são executados de forma independente, cada serviço pode ser atualizado, implantado e escalado para atender a demanda de funções específicas de um aplicativo.

Figura 16 – Microserviço.



Container: em vez de usar um sistema operacional para cada estrutura, como na virtualização, os Containers são blocos de espaços divididos pelo Docker em um servidor, o que possibilita a implementação de estruturas de Microserviços que compartilham o mesmo sistema operacional, porém, de forma limitada (conforme a demanda por capacidade). O fato de os Containers não terem seus próprios sistemas operacionais, permite que eles consumam menos recursos e, com isso, sejam mais leves.

Figura 17 – Containers.

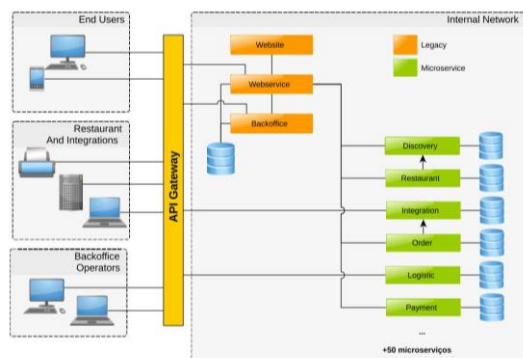


As ferramentas de orquestração de containers são aplicações em nuvem que permitem fazer o gerenciamento de múltiplos contêineres. Seus principais objetivos são:

- Cuidar do ciclo de vida dos containers de forma autônoma, subindo e distribuindo, conforme nossas especificações ou demandas.
- Gerenciar volumes e rede, que podem ser local ou no cloud provider de sua preferência.

O Kubernetes, ECS e o Docker são as principais plataformas de gerenciamento de contêineres. Dessa forma, essa é uma ferramenta para viabilizar a utilização de Containers e Microserviços em servidores com mais facilidade, pois permite empacotar os aplicativos para que possam ser movimentados facilmente. O Docker permite, por exemplo, que uma biblioteca possa ser instalada em diferentes Containers sem que haja qualquer interdependência entre eles. Essa característica tem o objetivo de facilitar o gerenciamento de códigos e aplicativos.

Figura 18 – Arquitetura de Microserviços.



Sistemas de Arquivos Distribuídos

Os sistemas de arquivos foram originalmente desenvolvidos como um recurso do S.O que fornece uma interface de programação conveniente para armazenamento em disco. São responsáveis pela organização, armazenamento, recuperação, atribuição de nomes, compartilhamento e proteção de arquivos. Projetados para armazenar e gerenciar muitos arquivos, com recursos para criação atribuição de nomes e exclusão.

Um sistema de arquivos distribuídos permite aos programas armazenarem e acessarem arquivos remotos exatamente como se fossem locais, possibilitando que os usuários acessem arquivos a partir de qualquer computador em uma rede.” (COULOURIS, et al., p. 284).

- Objetivo: permitir que os programas armazenem e acessem arquivos remotos exatamente como se fossem locais.
- Permitem que vários processos compartilhem dados por longos períodos, de modo seguro e confiável.
- O desempenho e segurança no acesso aos arquivos armazenados em um servidor devem ser compatíveis aos arquivos armazenados em discos locais.
- Os requisitos de um sistema de arquivos distribuídos são:

- Transparência.
- Atualização concorrente de arquivos.
- Replicação de arquivos.
- Heterogeneidade.
- Tolerância a falha.
- Consistência.
- Segurança.
- Eficiência.

Arquitetura do SAD

Modelo abstrato de arquitetura que serve para o Network File System (NFS) e Andrew File System (AFS).

Divisão de responsabilidades entre três módulos:

- Cliente.
- Serviço de arquivos planos.
- Serviço de diretórios.

Design aberto:

- Diferentes módulos cliente podem ser utilizados para implementar diferentes interfaces.
- Simulação de operações de arquivos de diferentes S.O.
- Otimização de performance para diferentes configurações de hardware de clientes e servidores.

Funções de um Sistema de Arquivos Distribuído:

- Armazenar e compartilhar programas e dados
 - Funções idênticas às de um sistema centralizado (local).
- Ênfase na disponibilidade, confiabilidade e segurança.
- Desempenho.
 - Questão importante porque acessos remotos podem ser significativamente mais lentos que os locais.
 - Não se pretende em geral que o SAD seja mais rápido que um SA local, mas sim que a degradação seja aceitável.

Sistemas de arquivo distribuídos devem ser vistos pelos clientes como um sistema de arquivo local. A transparência é muito importante para seu bom funcionamento. É necessário um bom controle de concorrência no acesso. Cache é importante.

Apache Hadoop

Hadoop é uma plataforma de software de código aberto para o armazenamento e processamento distribuído de grandes conjuntos de dados, utilizando clusters de computadores com hardware commodity.

Os serviços do Hadoop fornecem armazenamento, processamento, acesso, governança, segurança e operações de Dados.

Algumas das razões para se usar Hadoop é a sua capacidade de armazenar, gerenciar e analisar grandes quantidades de dados estruturados e não estruturados de forma rápida, confiável, flexível e de baixo custo.

- Escalabilidade e desempenho – distribuídos no tratamento de dados local para cada nó em um cluster Hadoop, permitindo armazenar, gerenciar, processar e analisar dados em escala petabyte.
- Confiabilidade – clusters de computação de grande porte são propensos a falhas de nós individuais no cluster. Hadoop é fundamentalmente resistente – quando um nó falha de processamento, é redirecionado para os nós restantes no cluster e os dados são automaticamente re-replicados em preparação para falhas de nó futuras.
- Flexibilidade – ao contrário de sistemas de gerenciamento de banco de dados relacionais tradicionais, você não precisa de esquemas estruturados criados antes de armazenar dados. Você pode armazenar dados em qualquer formato, incluindo formatos semiestruturados ou não estruturados, e em seguida, analisar e aplicar esquema para os dados quando ler.
- Baixo custo – ao contrário de software proprietário, o Hadoop é open source e é executado em hardware commodity de baixo custo.

O **HDFS** (Hadoop Distributed File System) é um sistema de arquivos distribuído, projetado para armazenar arquivos muito grandes, com padrão de acesso aos dados streaming, utilizando clusters de servidores facilmente encontrados no mercado e de baixo ou médio custo.

Não deve ser utilizado para aplicações que precisem de acesso rápido a um determinado registro e sim para aplicações nas quais é necessário ler uma quantidade muito grande de dados.

Outra questão que deve ser observada é que não deve ser utilizado para ler muitos arquivos pequenos, tendo em vista o overhead de memória envolvido.

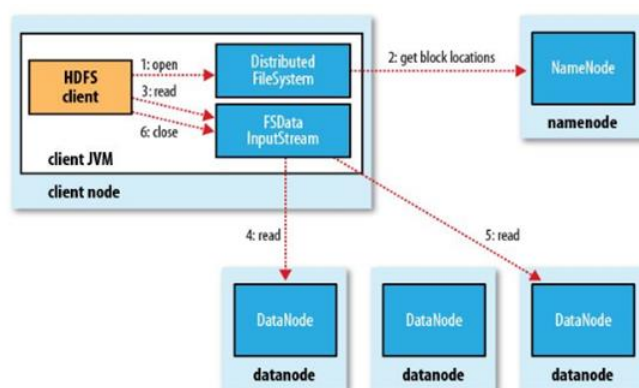
O HDFS tem 2 tipos de Nós: Master (ou Namenode) e Worker (ou Datanode).

O **Master** armazena informações da distribuição de arquivos e metadados.

Já o **Worker** armazena os dados propriamente ditos. Logo o Master precisa sempre estar disponível. Para garantir a disponibilidade podemos ter um backup (similar ao Cold Failover) ou termos um Master Secundário em um outro servidor. Nessa segunda opção, em caso de falha do primário, o secundário pode assumir o controle muito rapidamente.

Tal como um sistema Unix, é possível utilizar o HDFS via linha de comando.

Figura 18 – HDFS.

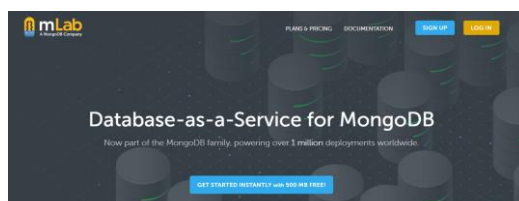


MongoDB na Nuvem

Conhecido um tempo atrás como MongoLab, o mLab é um serviço de banco de dados gerenciável que hospeda na nuvem um banco de dados MongoDB e é executado em provedores como a Amazon Web Services (AWS), Google Cloud e Microsoft Azure.

A parte mais interessante é que o serviço tem um plano gratuito que oferece 0,5 Gb para armazenamento de dados.

Figura 19 – Mongo.



Referências

ACID. Disponível em: <<http://blog.lucasrenan.com/propriedades-acid/>>. Acesso em: 12 fev. 2021.

AGREGAÇÃO. Disponível em: <<https://felipetoscano.com.br/agregacao-basica-no-mongodb/>>. Acesso em: 12 fev. 2021.

BANCO DE DADOS. Disponível em: <<https://blog.grancursosonline.com.br/o-que-e-um-banco-de-dados/>>. Acesso em: 12 fev. 2021.

BANCO NOSQL. Disponível em: <<https://danielteofilo.files.wordpress.com/2019/09/banco-de-dados-nosql-aula-03.pdf>>. Acesso em: 12 fev. 2021.

CONCEITOS FUNDAMENTAIS. Disponível em: <<https://www.devmedia.com.br/conceitos-fundamentais-de-banco-de-dados/1649>>. Acesso em: 12 fev. 2021.

CRUD Básico com MongoDB. Disponível em: <<https://www.mundojs.com.br/2020/03/17/crud-basico-com-mongodb/>>. Acesso em: 12 fev. 2021.

DATA INTEGRATION. In: Wikipédia, a enciclopédia livre. Flórida: Wikimedia Foudation, 2020. Disponível em: <https://en.wikipedia.org/wiki/Data_integration#cite_note-refone-1>. Acesso em: 12 fev. 2021.

DATABASE. Disponível em: <<https://medium.com/analytics-vidhya/no-sql-databases-an-introduction-eb9706fbc3>>. Acesso em: 12 fev. 2021.

DATE, C. J. Introdução a Sistemas de Banco de Dados. Elsevier Editora, 2004.

DBAAS. Disponível em: <<https://blogbrasil.westcon.com/o-que-e-banco-de-dados-como-servico-dbaas>>. Acesso em: 12 fev. 2021.

DESNORMALIZAÇÃO.

Disponível

em:

<<https://www.devmedia.com.br/desnormalizacao-de-bancos-de-dados/38623>>.

Acesso em: 12 fev. 2021.

DEVMEDIA. Disponível em: <<http://www.devmedia.com.br/>>. Acesso em: 12 fev. 2021.

ELMASRI, R.; NAVATHE, S. B. Sistemas de Banco de Dados. 4a ed., Pearson-Addison-Wesley, 2005.

ESQUEMA. Disponível em: <<https://www.lucidchart.com/pages/pt/o-que-e-um-esquema-de-banco-de-dados>>. Acesso em: 12 fev. 2021.

INACIO, Aylton. Criando e alimentando um cubo OLAP físico no MySQL. AyltonInacio Blog, 2018. Disponível em: <<https://ayltoninacio.com.br/blog/criando-e-alimentando-um-cubo-olap-fisico-no-mysql>>. Acesso em: 12 fev. 2021.

MODELAGEM DE DADOS. Disponível em: <<https://blog.academaiain1.com.br/o-que-e-e-gual-a-importancia-de-aprender-sobre-modelagem-de-dados/>>. Acesso em: 12 fev. 2021.

MONGODB. Disponível em: <<http://lucasmaiaesilva.com.br/posts/mongodb-opera%C3%A7%C3%B5es-de-crud/>>. Acesso em: 12 fev. 2021.

MONGOSHEL. Disponível em: <<https://medium.com/danieldiasjava/mongo-shell-749e21fe62c3>>. Acesso em: 12 fev. 2021.

NOSQL DATABASE. Disponível em: <<https://medium.com/swlh/4-types-of-nosql-databases-d88ad21f7d3b>>. Acesso em: 12 fev. 2021.

NOSQL. Disponível em: <<https://imasters.com.br/banco-de-dados/mongodb-para-iniciantes-em-nosql-parte-02>>. Acesso em: 12 fev. 2021.

OPERADORES LÓGICOS. Disponível em: <<https://pt.slideshare.net/gscavassa/6-operadores-de-comparao-e-lgicos-no-mongodb>>. Acesso em: 12 fev. 2021.

RACKSPACE. Disponível em: <<https://www.rackspace.com/pt-br/library/>>. Acesso em: 12 fev. 2021.

RELATIONAL DATABASE. Disponível em: <<https://www.oracle.com/br/database/what-is-a-relational-database/>>. Acesso em: 12 fev. 2021.

SGBD. Disponível em: <<https://dicasdeprogramacao.com.br/o-que-e-um-sgbd/>>. Acesso em: 12 fev. 2021.

SISTEMA DE ARQUIVOS. Disponível em: <<https://www.techtudo.com.br/dicas-e-tutoriais/noticia/2016/02/entenda-o-que-e-sistema-de-arquivos-e-sua-utilidade-no-pc-e-no-celular.html>>. Acesso em: 12 fev. 2021.

VLAD V, WENDY A. NEU, ANDY O, SAM A. An Introduction to Cloud Databases. Released November 2019. Publisher(s): O'Reilly Media, Inc.