Ministério da Justiça

Termo de Cooperação/Projeto:

Acordo de Cooperação Técnica FUB/CDT e MJ/SE Registro de Identidade Civil – Replanejamento e Novo Projeto Piloto

Documento:

RT de Infraestrutura Tecnológica: Banco de Dados SQL, NewSQL e Armazenadores NoSQL

Data de Emissão:

31/03/2015

Elaborado por:

Universidade de Brasília – UnB
Centro de Apoio ao Desenvolvimento Tecnológico – CDT
Laboratório de Tecnologias da Tomada de Decisão – LATITUDE.UnB

MINISTÉRIO DA JUSTIÇA

José Eduardo Cardozo
Ministro

Marivaldo de Castro Pereira
Secretário Executivo

Helvio Pereira Peixoto
Coordenador Suplente do Comitê Gestor do SINRIC

UNIVERSIDADE DE BRASÍLIA

Ivan Marques Toledo Camargo
Reitor

Paulo Anselmo Ziani Suarez
Diretor do Centro de Apoio ao Desenvolvimento Tecnológico – CDT

Rafael Timóteo de Sousa Júnior
Coordenador do Laboratório de Tecnologias da Tomada de Decisão – LATITUDE

EQUIPE TÉCNICA
Ana Maria da Consolação Gomes Lindgren
Andréa Benoliel de Lima
Celso Pereira Salgado
Delluiz Simões de Brito
Elaine Fabiano Tocantins
Fernando Saliba Oliveira
Fernando Teodoro Filho
Guilherme Braz Carneiro
Joaquim de Oliveira Machado
José Alberto Sousa Torres
Marcelo Martins Villar
Raphael Fernandes de Magalhães Pimenta
Rodrigo Borges Nogueira
Rodrigo Gurgel Fernandes Távora
Sara Lais Rahal Lenharo

EQUIPE TÉCNICA
Flávio Elias Gomes de Deus (Pesquisador Sênior)
William Ferreira Giozza (Pesquisador Sênior)
Ademir Agustinho de Rezende Lourenço
Adriana Nunes Pinheiro
Alysson Fernandes de Chantal
Amanda Almeida Paiva
Andréia Campos Santana
Antônio Claudio Pimenta Ribeiro
Carolinne Januária de Souza Martins
Caio Rondon Botelo de Carvalho
Daniela Carina Pena Pascual
Danielle Ramos da Silva
Diogenes Ferreira Reis Fustinoni
Eduarda Simões Veloso Freire
Fábio Lúcio Lopes Mendonça
Fábio Mesquita Buiati
Glaidson Menegazzo Verzeletti
Heverson Soares de Brito
Johnatan Santos de Oliveira
José Carneiro da Cunha Oliveira Neto
José Elenilson Cruz
Kelly Santos de Oliveira Bezerra
Luciano Pereira dos Anjos
Luciene Pereira de Cerqueira Kaipper
Luiz Antônio de Souto Evaristo
Luiz Claudio Ferreira
Marcos Vinicius Vieira da Silva
Marco Schaffer
Mirele Maria Cavalcante Rocha
Pedro Augusto Oliveira de Paula
Renata Elisa Medeiros Jordão
Roberto Mariano de Oliveira Soares
Sandro Augusto Pavlik Haddad
Sergio Luiz Teixeira Camargo
Soleni Guimarães Alves
Suzane Lais De Freitas
Valério Aymoré Martins
Vera Lopes de Assis
Vinicius de Morais Alves
Wladimir Rodrigues da Fonseca

HISTÓRICO DE REVISÕES

Data	Versão	Descrição
16/07/2014	0.1	Versão inicial.
21/07/2014	0.2	Versão incorporando introdução
12/08/2014	0.3	Versão incorporando Armazenadores Orientados a Colunas e a Grafos
01/12/2014	0.4	Versão melhorada pela Equipe Técnica
31/03/2015	0.5	Versão final para revisão

latitude
Laboratório de tecnologias da tomada de decisão

Universidade de Brasília – UnB
Campus Universitário Darcy Ribeiro - FT – ENE – Latitude
CEP 70.910-900 – Brasília-DF
Tel.: +55 61 3107-5598 – Fax: +55 61 3107-5590

SUMÁRIO

1 INTRODUÇÃO 6
2 ARMAZENAMENTO DE DADOS 8
2.1 Introdução 8
2.2 Modelos de Armazenamento e SGBDs 9
2.3 Principais características dos armazenadores NoSQL 15
3 ELEMENTOS DO PROCESSAMENTO MASSIVO DE DADOS 19
3.1 Introdução 19
3.2 *BigData* 20
3.3 HDFS - Sistema de Arquivos Distribuídos *Hadoop* 22
3.4 *MapReduce* 23
3.5 Ecossistema *Hadoop* 24
4 DIMENSÕES ENVOLVIDAS NO CONCEITO DE NoSQL 26
4.1 Consistência Eventual ou Relaxada 26
4.2 Modelo de Dados Sem Esquema (Esquema Flexível) 28
4.3 Índices 29
4.4 Compressão 29
4.5 Escalabilidade e Disponibilidade pela Distribuição 29
4.5.1 Particionamento Horizontal: *Sharding* 31
4.5.2 Particionamento Vertical: Armazenamento em Colunas 32
4.6 Consistência Histórica (Versionamento de Dados) 34
4.7 Persistência Poliglota 34
5 ARMAZENADORES NOSQL: TIPOLOGIA NOTÓRIA 35
5.1 Armazenamento Chave-Valor (*Key-Value Stores*) 37
5.2 Armazenamento Orientado a Colunas (*BigTable-style Databases*) 38
5.3 Armazenamento Orientado a Documentos (*Document Databases*) 39
5.4 Armazenamento Orientado a Grafos (*Graph Databases*) 39
6 CHAVE-VALOR (*KEY VALUE*) 41
6.1 Introdução 41
6.2 Quando Usar 42
6.3 Quando Não Usar 42
7 ORIENTADO A COLUNAS (*COLUMN-STORES*) 43
7.1 Introdução 43
7.1.1 *Auto-Sharding* 44

7.1.2 Versionamento 45
7.1.3 Compressão de Dados 45
7.2 Quando Usar 46
7.3 Quando Não Usar 46
8 ORIENTADO A DOCUMENTOS (*DOCUMENT-STORES*) 47
8.1 Introdução: Documentos e Mensagens 47
8.2 Quando Usar 49
8.3 Quando Não Usar 49
9 ORIENTADO A GRAFOS (*GRAPH-STORES*) 50
9.1 Introdução: Propriedades dos Grafos 50
9.2 Quando usar 52
9.3 Quando Não Usar 52
10 SEGURANÇA: CONFIDENCIALIDADE E PRIVACIDADE 53
10.1 Segurança no Armazenamento 53
10.2 Probabilístico (*Random*) 53
10.3 Pseudônimo (Determinístico) 53
10.4 Busca por Palavras (*Searchwords*) 53
10.5 Busca Probabilística por Palavras (*Probabilistic Searchwords*) 54
10.6 Criptografia Homomórfica 54
10.7 Criptografia de Preservação de Ordem 54
11 CONCLUSÃO 56
REFERÊNCIAS BIBLIOGRÁFICAS 57

1 INTRODUÇÃO

A Secretaria Executiva (SE/MJ), vinculada ao Ministério da Justiça (MJ), é responsável por viabilizar o desenvolvimento e a implantação do Registro de Identidade Civil, instituído pela Lei nº 9.454, de 7 de abril de 1997, regulamentado pelo Decreto nº 7.166, de 5 de maio de 2010.

Atualmente, a República Federativa do Brasil conta com sistema de identificação de seus cidadãos amparado pela Lei nº 7.116, de 29 de agosto de 1983. Essa lei assegura validade nacional às Carteiras de Identidade, ou Cédulas de Identidade; confere também autonomia gerencial às Unidades Federativas no que concerne à expedição e controle dos números de registros gerais emitidos para cada documento. Essa condição de autonomia, ao contrário do que pode parecer, fragiliza o sistema de identificação, já que dá condições ao cidadão de requerer legalmente até 27 (vinte e sete) cédulas de identidades diferentes. Com essa facilidade legal, inúmeras possibilidades fraudulentas se apresentam de maneira silenciosa, pois, na grande maioria dos casos, os Institutos de Identificação das Unidades Federativas não dispõem de protocolos e aparato tecnológico para identificar as duplicações de registro vindas de outros estados, ou até mesmo do seu próprio arquivo datiloscópico. Consoante aos fatos, os Institutos de Identificação não trabalham interativamente para que haja trocas de informações de dados e geração de conhecimento para manuseio inteligente e seguro para individualização do cidadão em prol da sociedade.

Com foco na busca de soluções para tais problemas, o Projeto RIC prevê a administração central dos dados biográficos e biométricos dos cidadãos no Cadastro Nacional de Registro de Identificação Civil (CANRIC) e ABIS (do inglês *Automated Biometric Identification System*), respectivamente. A previsão desse novo modelo sustenta a não duplicação de registros e a consequente identificação unívoca dos cidadãos brasileiros natos e naturalizados. O Projeto RIC, portanto, visa otimizar o sistema de identificação e individualização do cidadão brasileiro nato e naturalizado com vistas a um perfeito funcionamento da gestão de dados da sociedade, agregando valor à cidadania, à gestão administrativa, à simplificação do acesso aos serviços disponíveis ao cidadão e à segurança pública do país.

Nesse contexto, o termo de cooperação entre MJ/SE e FUB/CDT define um projeto que objetiva identificar, mapear e desenvolver parte dos processos e da infraestrutura tecnológica necessária para viabilizar a implantação do número único de Registro de

Identidade Civil – RIC no Brasil.

Resultante de um subconjunto das atividades previstas para inicialização da cooperação MJ/SE e FUB/CDT, o presente documento contempla uma primeira visão da infraestrutura tecnológica no armazenamento de dados, tratando de apresentar os modelos de banco de dados tradicionais, os modelos de banco de dados relacionais distribuídos e os modernos modelos de armazenamento de dados NoSQL, apontando vantagens e desvantagens associados aos prováveis usos e às aplicações não aconselháveis.

2 ARMAZENAMENTO DE DADOS

No que se refere aos serviços de armazenamento de dados para o estabelecimento de uma infraestrutura com condições de suportar o esperado volume de acesso massivo, contínuo e distribuído baseado em serviços *web* - como idealizado para o RIC - é necessário que se discuta os problemas notórios envolvidos no armazenamento de dados, principalmente com o uso mais recente de técnicas que permitam o tratamento massivo de dados. Essas técnicas estão ora concentradas no estudo dos assuntos de *BigData* e de armazenadores NoSQL.

2.1 Introdução

Bancos de dados relacionais e seus modelos de gerenciamento baseados em Sistemas Gerenciadores de Banco de Dados (SGBDs) têm sido uma tecnologia de sucesso usada há vinte anos para fornecer persistência, controle de concorrência e um mecanismo de integração entre aplicações.

Neste sentido, ainda hoje, o uso de SGBDs tem tipicamente um papel fundamental na concepção e implementação de aplicações de negócios, em que ocorre a necessidade de se manter / persistir informações sobre seus usuários, produtos, sessões, cadastramento e assim por diante, por meio de algum *backend* de armazenamento SGBD que propicie uma camada de persistência para a aplicação cliente (*frontend*).

Isso funciona bem para um número limitado de registros, mas com o aumento dramático dos dados persistidos, alguns dos detalhes arquitetônicos da implementação dos SGBDs usuais mostram sinais de fraqueza, a saber.

- SGBDs usam uma abordagem de normalização com base em tabela de dados, e esse modelo é limitado. Certas estruturas de dados não podem ser representadas sem adulteração deles, dos programas, ou de ambos.
- O desempenho também é perdido quando os bancos de dados relacionais normalizam os dados, pois a normalização pede mais tabelas, união (*join*) de tabelas, chaves e índices e, dessa maneira, mais operações internas do banco de dados para implementar as consultas. Assim, se o banco de dados começa a crescer para o tamanho de *terabytes*, o comportamento se torna ineficiente (lento).

- A garantia de integridade, imposta pelas leis de Codd para SGBDs, efetua inequivocamente uma série de testes de integridade que adiam o processo de persistência, principalmente nas operações de inclusão, alteração e exclusão de registros. Ainda na execução sucessiva de operações de inclusão, alteração e exclusão, para garantir a consistência e integridade, em diversas situações, o tratamento dessas operações não pode ser paralelizado.
- Os desenvolvedores de aplicativos notam certa frustração com a diferença de impedância entre o modelo relacional e as estruturas de dados que estão acontecendo em memória. Neste sentido, há um movimento atual de usar bancos de dados como pontos de integração para encapsular os bancos de dados em aplicativos e integração através de serviços.
- Dada a necessidade de dar garantia do princípio de disponibilidade no suporte a grandes volumes de dados, em geral, essa execução se dá através de *clusters* de computadores. Porém, dificuldades são encontradas em bancos de dados relacionais em *cluster*, pois esses não são projetados para funcionar de forma eficiente em *clusters*.
- SGBDs em geral não permitem versionamento de dados. Em determinadas situações, as atualizações não são permitidas porque elas destroem a informação. Em vez disso, quando os dados mudam, o banco de dados apenas adiciona outro registro, o que aumenta o tamanho de dados a serem percorridos.

Para entendimento dos mecanismos propostos para solução desses problemas, é necessária uma análise histórica dos conceitos e produtos afetos, principalmente do processamento massivo de dados através da aplicação de soluções comuns associadas a distribuição em redes e sistemas.

2.2 Modelos de Armazenamento e SGBDs

Os primeiros Sistemas Gerenciadores de Banco de Dados (SGBD) surgiram por volta de 1960 e foram desenvolvidos com base nos primitivos sistemas de arquivos. Dentre as principais características de um SGBD destaca-se o controle de concorrência, a segurança, a recuperação de falhas, os mecanismos de gerenciamento de armazenamento de dados, e o controle das restrições de integridade da base de dados.

Outra importante função de um SGBD é o gerenciamento de transações. Uma

transação pode ser definida como uma coleção de operações que desempenha uma função lógica dentro de uma aplicação do sistema de banco de dados. Em outras palavras, uma transação representa um conjunto de operações de leitura ou escrita que são realizadas no banco de dados. A execução de transações em um SGBD deve obedecer a algumas propriedades a fim de garantir o correto funcionamento do sistema e a respectiva consistência dos dados. Estas propriedades são chamadas de ACID (um acrônimo dessas propriedades), e que são definidas a seguir [1].

• Atomicidade: todas as operações da transação devem ser executadas, ou seja, ou a transação é executada por completo, ou nada é executado (ela é tratada como atômica). Assim, se uma parte da transação falhar, toda transação falhará, e é responsabilidade de um subsistema de restauração de transações do SGBD garantir a atomicidade: se durante qualquer transação ocorrer uma falha na unidade de trabalho, a transação deve ser desfeita (*rollback*); ou quando todas as ações são efetuadas com sucesso, a transação é efetivada (*commit*).

• Consistência: após uma transação ter sido concluída, o banco de dados deve permanecer em um estado consistente, ou seja, deve satisfazer as condições de consistência e restrições de integridade previamente assumidas. Assim, uma transação preservará a consistência se a sua execução completa fizer o banco de dados passar de um estado consistente para o outro, isto é, garante que todos os dados serão escritos no banco de dados. Elmasri & Navathe [1] definem que um estado do banco de dados é a coleção de todos os itens de dados armazenados (valores) no banco de dados em um dado momento. Um estado consistente do banco de dados satisfaz as restrições especificadas no esquema, bem como quaisquer outras restrições que foram antecipadamente impostas.

• Isolamento: se duas transações estão sendo executadas concorrentemente, seus efeitos devem ser isolados uma da outra. Esta propriedade está relacionada ao controle de concorrência do SGBD, o que permite que duas ou mais pessoas acessem uma mesma base de dados ao mesmo tempo, e o sistema executa o controle para que um acesso não interfira no outro. Significa que a execução de uma transação não deve sofrer interferência de quaisquer outras transações concorrentes.

• Durabilidade: uma vez que uma transação ocorreu com sucesso, seu efeito não poderá mais ser desfeito, mesmo em caso de falha. Esta propriedade está relacionada a capacidade de recuperação de falhas do SGBD, pois as mudanças aplicadas ao banco de

dados por uma transição efetivada (*commit*) devem permanecer no banco de dados. Essas mudanças não devem ser perdidas em razão de nenhuma falha.

Diferentes modelos de dados foram propostos desde o surgimento dos SGBDs, os quais se diferenciam pelos conceitos adotados para representação dos dados do mundo real. Inicialmente, foram propostos os modelos hierárquicos e de rede.

O modelo hierárquico foi o modelo pioneiro no projeto de banco de dados. Os dados são organizados em hierarquias ou árvores, sendo cada nó da árvore uma coleção de atributos. O registro da hierarquia que precede a outros é o registro-pai, os outros são chamados de registros-filhos. Uma ligação é uma associação que é feita apenas entre dois registros. O relacionamento entre um registro-pai e vários registros-filhos possui cardinalidade 1:N [2]. Os dados organizados segundo este modelo podem ser acessados segundo uma sequência hierárquica com uma navegação *top-down*, conforme ilustra a Figura 1.

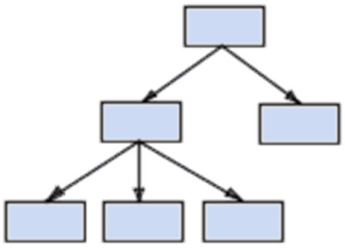


Figura 1 - Modelo de dados Hierárquico (pt.kioskea.net).

O modelo em rede foi proposto como uma extensão ao modelo hierárquico, no qual não existe o conceito de hierarquia e um mesmo registro pode estar envolvido em várias associações. Assim, permite que haja várias associações entre dois registros. Tem a sua estrutura em forma de grafos. Ao contrário do modelo hierárquico, em que qualquer acesso aos dados passa pela raiz, o modelo em rede possibilita acesso a qualquer nó da rede sem passar pela raiz (Figura 2). Para esses dois modelos, qualquer acesso à base de dados – inserção, consulta, alteração ou remoção – é feito em um registro de cada vez [2].

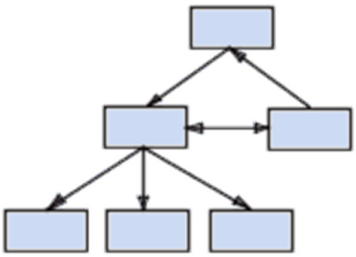


Figura 2 - Modelo dados em Rede (pt.kioskea.net)**.**

No início dos anos 70, surgiram os bancos de dados relacionais, os quais se firmaram como solução comercial para armazenamento e gerenciamento de dados convencionais, ou seja, dados que possuem uma estrutura fixa, bem definida e com tipos de dados simples, como os dados gerados e manipulados por aplicações convencionais de bancos de dados (ex.: sistemas de controle de estoque e folha de pagamento). Esse modelo foi criado pela necessidade de aumentar a independência de dados nos sistemas gerenciadores de banco de dados, e para prover um conjunto de funções apoiadas em álgebra relacional para armazenamento e recuperação de dados [2].

Tendo por base a teoria dos conjuntos e a álgebra relacional, o modelo relacional é organizado em forma de tabelas (Figura 3). Na nomenclatura do modelo relacional formal, uma linha é chamada tupla, um cabeçalho de coluna é um atributo e a tabela é conhecida como relação [1]. Cada tupla representa uma coleção de valores de dados relacionados que correspondem a uma entidade ou relacionamento do mundo real. A maior vantagem do modelo relacional sobre seus antecessores é a representação simples dos dados e a facilidade com que consultas complexas podem ser expressas. Tem como linguagem padrão para criação, manipulação e consultas a linguagem SQL (*Structured Query Language*).

O objetivo desse modelo é representar os dados de forma mais simples, por meio de um de conjuntos de tabelas inter-relacionadas. Este modelo abandona os conceitos anteriores, tornando os bancos de dados mais flexíveis, tanto na forma de representar as relações entre os dados, como na tarefa de modificação de sua estrutura, sem ter que reconstruir todo o banco de dados.

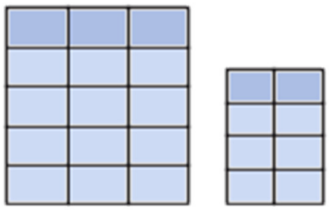


Figura 3 - Modelo de dados Relacionais (pt.kioskea.net).

De uma maneira geral, a simplicidade do modelo relacional contribuiu para sua grande disseminação e adoção. Porém, à medida que as aplicações de bancos de dados foram evoluindo, surgiu a necessidade de manipulação de outros formatos de dados, como imagem, som e vídeo, bem como de tipos de dados complexos. A fim de atender aos requisitos destas aplicações de bancos de dados não convencionais, novas soluções foram propostas, como os bancos de dados orientados a objetos (BDOO) e os bancos de dados objeto-relacionais (BDOR).

O modelo Orientado a Objetos surgiu em meados de 1980 para armazenamento de dados complexos, não adequados aos sistemas relacionais. Foram inspirados nas práticas de programação orientada a objetos e incorporaram algumas de suas funcionalidades como encapsulamento de operações, herança de objetos e registros de dados abstratos, já difundidos em linguagens de programação como o *SmallTalk* e o C++. Assim, os dados são armazenados sob a forma de objetos e só podem ser manipulados por métodos definidos nas classes das quais pertencem [2]. Esses objetos podem conter referências para outros objetos, e as aplicações podem acessar os dados requeridos usando um estilo de navegação que contenha um conjunto de linguagem de programação orientada a objetos.

Com seu objetivo principal é tratar os tipos de dados complexos como um tipo abstrato (objeto), a filosofia do modelo de dados orientado a objetos consiste em agrupar os dados e o código que manipula estes dados em um único objeto, estruturando-os de forma que possam ser agrupados em classes. Isso significa que os objetos de banco de dados agrupados podem usar o mesmo mecanismo de herança para definir superclasses e subclasses de objetos, criando assim hierarquias (Figura 4).

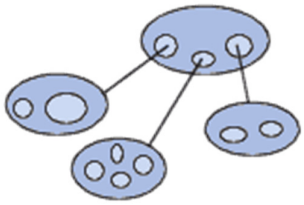


Figura 4 - Modelo de dados Orientado a Objetos (pt.kioskea.net).

Os Sistemas Objetos-Relacionais são bancos de dados relacionais que adicionaram a seus produtos capacidade de incorporar tipos de dados mais complexos além de recursos de orientação a objetos.

No entanto, isso não os torna sistemas puramente orientados a objetos, apesar de sua denominação ORDMS - *Object-Relacional Database Management System* (Sistema de Gerenciamento de Banco de Dados Objeto-Relacional). Esses sistemas na realidade implementam uma camada de abstração de dados em cima de métodos relacionais, o que torna possível a manipulação de dados mais complexos. Seguem, portanto, as especificações do SQL3, as quais fornecem capacidades estendidas e de objetos adicionadas ao padrão SQL. Todas as características relacionais permanecem, ou seja, as tabelas continuam a existir, porém elas possuem alguns recursos adicionais.

Posteriormente, com o surgimento da Web, outras aplicações de banco de dados começaram a ser desenvolvidas, originando, dessa forma, novos requisitos de bancos de dados. Dentre estes requisitos destaca-se a necessidade de manipulação de grandes volumes de dados não estruturados ou semiestruturados, bem como novas necessidades de disponibilidade e escalabilidade. Assim, a fim de atender aos requisitos destas aplicações, novas soluções para gerenciamento de dados começaram a ser propostas.

Neste contexto, surgiu uma nova categoria de Banco de Dados, a qual foi denominada NoSQL (*"Not Only SQL"*), um termo cunhado por Eric Evans, em resposta a uma pergunta de Johan Oskarsson, que estava tentando encontrar um nome para o evento de um novo espaço do sistema de armazenamento de dados muito emergentes. NoSQL é um neologismo acidental. Não há definição prescritiva, mas uma observação de características comuns. Esse termo foi proposto com o objetivo de atender aos requisitos de gerenciamento de grandes volumes de dados, semiestruturados ou não estruturados, os quais necessitam de alta disponibilidade e escalabilidade. Essa necessidade de uma nova tecnologia de BD surgiu como consequência da ineficiência dos bancos de dados relacionais em lidar com esta tarefa. Assim, o termo NoSQL faz referência a SGBDs que

não adotam o modelo relacional e são mais flexíveis quanto às propriedades ACID. Esta flexibilidade torna-se necessária devido aos requisitos de alta escalabilidade necessários para gerenciar grandes quantidades de dados, bem como para garantir a alta disponibilidade dos mesmos.

Os bancos de dados NoSQL têm sido amplamente adotados em empresas como Facebook, Amazon e Google com o intuito de atender às suas demandas de escalabilidade, alta disponibilidade e dados não estruturados. Além disso, atualmente, diversos bancos de dados NoSQL de código livre estão disponíveis, como: *HBase*, *Cassandra*, *Hypertable*, *MongoDB*, *Redis*, *CouchDB* e *Dynamo6*.

Tendo em vista o crescente interesse na adoção desta tecnologia, bem como os novos desafios gerados pelo uso do NoSQL, torna-se fundamental o conhecimento de seus principais conceitos e sua utilização. Em particular, estes conhecimentos são de grande relevância para organizações que têm interesse em aplicações *Web* colaborativas, as quais, na maioria das vezes, necessitam de uma tecnologia que ofereça de maneira simples e eficiente o suporte ao gerenciamento e escalabilidade de grandes volumes de dados.

2.3 Principais características dos armazenadores NoSQL

Os armazenadores NoSQL apresentam algumas características fundamentais, as quais os diferenciam dos tradicionais sistemas de bancos de dados relacionais, tornando-os adequados para armazenamento de grandes volumes de dados não estruturados ou semiestruturados. A seguir, descrevemos alguma destas características.

• Escalabilidade horizontal: à medida que o volume de dados cresce, aumenta a necessidade de escalabilidade e melhoria de desempenho. Dentre as soluções para este problema, temos a escalabilidade vertical, a qual consiste em aumentar o poder de processamento e armazenamento das máquinas, e a escalabilidade horizontal, na qual ocorre um aumento no número de máquinas disponíveis para o armazenamento e processamento de dados. A escalabilidade horizontal tende a ser uma solução mais viável, porém requer que diversos *threads*/processos de uma tarefa sejam criados e distribuídos. Neste caso, o uso de um banco de dados relacional poderia ser inviável, uma vez que diversos processos conectando simultaneamente um mesmo conjunto de dados causaria uma alta concorrência, aumentando, consequentemente, o tempo de acesso às tabelas envolvidas. Assim, a ausência de bloqueios (que sequencia as tarefas) é uma característica

fundamental dos bancos de dados NoSQL, permitindo a escalabilidade horizontal e tornando esta tecnologia adequada para solucionar os problemas de gerenciamento de volumes de dados que crescem exponencialmente. Uma alternativa muito conhecida para alcançar escalabilidade horizontal é o *Sharding*, que consiste em dividir os dados em múltiplas tabelas a serem armazenadas ao longo de diversos nós de uma rede. Neste caso, as aplicações têm que resolver a complexidade gerada pela partição de informações como, por exemplo, a execução de *joins* e outros comandos. Fazer *sharding*, em um contexto de banco de dados relacionais, de forma manual não é uma tarefa.

• Consistência eventual: é uma característica de bancos NoSQL relacionada ao fato de que, no processamento e armazenamento de dados, a consistência nem sempre pode ser mantida entre os diversos pontos de distribuição de dados. Esta característica tem como princípio o teorema de consistência, disponibilidade e tolerância ao particionamento - CAP (*Consistency, Availability e Partition tolerence*), e preconiza que, em um dado momento, só é possível garantir duas dessas três propriedades. No contexto de armazenamento de dados na *Web*, por exemplo, geralmente são privilegiadas a disponibilidade e a tolerância ao particionamento. Assim, em armazenadores NoSQL, tem-se outro conjunto de conceitos denominado Disponibilidade básica, Estado leve e Consistência eventual - BASE (*Basic Availability, Soft state, Eventual consistency*). Neste modelo, o sistema armazenador é planejado para tolerar inconsistências temporárias (consistências eventuais) a fim de poder priorizar disponibilidade. O Quadro 1 compara esses princípios:

ACID	BASE
Consistência forte	Fraca consistência
Isolamento	Foco em Disponibilidade
Concentra-se em "commit"	Melhor esforço
Transações aninhadas	Respostas aproximadas
Disponibilidade	Mais simples e mais rápido
Conservador (pessimista)	Agressivo (otimista)
Evolução difícil (por exemplo, esquema)	Evolução mais fácil

Quadro 1 – Comparação dos princípios de ACID e BASE

• Ausência de esquema ou esquema flexível: uma característica comum dos bancos de dados NoSQL é a ausência completa ou quase total do esquema que define a estrutura dos dados modelados. Esta ausência de esquema facilita tanto a escalabilidade quanto contribui para um maior aumento da disponibilidade. Em contrapartida, não há garantias da integridade dos dados, o que ocorre nos bancos relacionais, devido à sua estrutura rígida.

Estas estruturas são, em sua maioria, baseadas em um conceito de chave-valor, permitindo uma alta flexibilidade na forma como os dados são organizados.

• Suporte nativo a replicação: outra forma de prover escalabilidade é por meio da replicação. Permitir a replicação de forma nativa diminui o tempo gasto para recuperar informações. Existem duas abordagens principais para replicação, a saber.

- *Master-Slave* (Mestre-Escravo): cada escrita no banco resulta em N escritas no total, onde N é o número de nós escravos. Nesta arquitetura a escrita é feita no nó mestre, sendo a escrita refeita em cada nó escravo pelo nó mestre. A leitura torna-se mais rápida, porém a capacidade de escrita torna-se um gargalo nesta abordagem, assim, não é recomendada quando se tem um grande volume de dados.
- *Multi-Master*: admite um modelo no qual temos vários nós mestres, de forma que é possível diminuir o gargalo gerado pela escrita que ocorre na abordagem mestre-escravo. Porém, a existência de diversos nós mestres pode causar um problema de conflito de dados.

Estas formas, principalmente a forma *Master-Slave*, são providas nativamente e são a base de distribuição dos modelos de armazenamento NoSQL.

• API simples para acesso aos dados: como o objetivo da solução NoSQL é prover uma forma eficiente de acesso aos dados, oferecendo alta disponibilidade e escalabilidade, o foco não está em como os dados são armazenados e sim como poderemos recuperá-los de forma eficiente. Para isto, é necessário que APIs sejam desenvolvidas para facilitar o acesso a estas informações, permitindo que qualquer aplicação possa utilizar os dados do banco de forma rápida e eficiente. Para extensão do conhecimento nessas características citadas é importante o conhecimento de algumas técnicas importantes para a implementação das funcionalidades do NoSQL, a saber.

- *Map/Reduce*: é a técnica que dá suporte ao manuseio de grandes volumes de dados distribuídos ao longo dos nós de uma rede. Na fase de *Map*, os problemas são quebrados em subproblemas, que são distribuídos em outros nós na rede. E na fase *Reduce*, os subproblemas são resolvidos em cada nó filho e o resultado é repassado ao pai, que, sendo ele também filho, repassaria ao seu pai, e assim por diante até chegar ao nó raiz.
- *Hashing*: é o mecanismo que suporta o armazenamento e recuperação em banco de

dados distribuídos, onde a quantidade de nós de armazenamento está em constante modificação. O uso desta técnica é interessante porque evita um grande número de migração de dados entre os nós de armazenamento, os quais podem ser alocados e desalocados para a distribuição dos dados.

- Controle de Concorrência Multiversão – MVCC (*Multiversion Concurrency Control*): é o mecanismo que dá suporte a transações paralelas em um banco de dados. Ao contrário do esquema clássico de gerenciamento de transações, em geral, os modelos NoSQL não fazem uso de *locks*, o que permite que operações de leitura e escrita sejam feitas simultaneamente. Esta arquitetura é, em geral, suportada por técnicas de Vector Clocks, as quais são usados para gerar uma ordenação dos eventos acontecidos em um sistema. Pela possibilidade de várias operações estarem acontecendo ao mesmo tempo sobre o mesmo item de dado distribuído, o uso de um log de operações com as suas respectivas datas é importante para determinar qual a versão de um determinado dado é a mais atual. Eles têm papel fundamental no estabelecimento de modelos de consistência eventual e cache adiantado dos armazenadores NoSQL, que serão tratados com mais detalhes nas discussões das dimensões (teorias e técnicas) envolvidas no armazenamento NoSQL.

3 ELEMENTOS DO PROCESSAMENTO MASSIVO DE DADOS

3.1 Introdução

Observou-se na última década um grande avanço na popularização da Internet bem como o aumento da quantidade e da complexidade dos serviços oferecidos. O processamento, armazenamento e segurança desses dados são fatores críticos que precisam ser avaliados, pois os mecanismos convencionais de gerenciamento de dados não oferecem o suporte adequado. Portanto, um dos grandes desafios computacionais da atualidade é armazenar, manipular e analisar, de forma inteligente, a grande quantidade de dados existente gerada por serviços e sistemas *Web*, redes sociais, entre outros, alcançando a dimensão de *petabytes* diários.

Vale observar que todo esse suporte tecnológico é fundamentado em sistemas distribuídos heterogêneos, suportados por protocolos de rede e comunicações, necessitando diretamente de aplicações de segurança da informação na manipulação, tramitação e disponibilização de dados e informações relacionadas.

Gigantescas fontes de dados têm se tornado um fator diferencial de informação de alto valor agregado. Entretanto, muitas empresas e corporações não sabem como aproveitar o real valor dessa informação. A maioria desses dados é armazenada em uma forma não estruturada e em sistemas, linguagens e formatos bem diferentes em ambientes distribuídos, e em muitos casos, incompatíveis entre si. A parte de prospecção de tecnologias e de segurança em sistemas de informação relacionada ao armazenamento, manipulação e análise de grandes massas de dados é fundamental para a consecução da produção de conhecimentos em novos ambientes de tecnologia da informação.

É importante observar que quando se considera o termo grande massa de dados, não se refere somente ao volume, mas também à variedade e à velocidade necessária para o seu processamento. Como existem diversos recursos geradores para esses dados, surge uma enorme variedade de formatos, alguns estruturados e outros não estruturados. Para geradores de dados estruturados, temos como exemplos sistemas corporativos e aplicações *Web*; já para os dados não estruturados, temos os *logs* de sistemas, as redes sociais, dispositivos móveis, imagens, vídeos, sensores. Por fim, também está relacionada à velocidade, pois muitas das novas aplicações precisam de respostas em um curto prazo e, em muitos casos, em tempo real. Agregue-se a tal realidade a ausência de processos de

segurança consagrados e a ausência de tecnologia adequada de suporte em sistemas distribuídos e temos um problema em escala volumétrica de segurança da informação. E este problema não pode ser ignorado e muito menos desconsiderado na geração de novas tecnologias e de novos conhecimentos em segurança da informação e tecnologias de comunicação envolvidas.

Assim, as empresas estão mais focadas em fornecer informações mais específicas, tais como recomendações ou anúncios *on-line*, sendo que sua capacidade de fazer isso influencia diretamente o seu sucesso como um negócio. Sistemas como *Hadoop* capacitam essas empresas a coletar e processar *petabytes* de dados, oriundos de diversas fontes de informação, sejam elas estruturadas ou sem nenhum esquema pré-definido.

No passado, a única opção para manter todos os dados coletados era a eliminação dos dados mais antigos, como por exemplo, reter apenas os últimos “n” dias. Embora esta seja uma abordagem viável a curto prazo, ela reduz a oportunidade de análise dos dados históricos.

No sentido de suportar a coleta e o armazenamento massivo de dados, foi necessário que se redefinisse os modelos de armazenamento físico em disco, para garantir uma distribuição eficiente desses dados entre diversos servidores bem como garantir a disponibilidade dos arquivos por balanceamento de cópias desses dados junto a esses servidores. A princípio, então, para entender-se o conceito de gerenciamento de dados distribuídos, se faz necessário o estudo dos modelos de armazenamento físico distribuído, tais como o *Hadoop Distributed File System* - HDFS, o *Google Fle System* – GFS e o *LexisNexis High-Performance Computing Cluster* – HPCC.

3.2 *BigData*

Uma solução proposta para o problema anteriormente mencionado é o *Apache Hadoop*[1], um *framework* para o armazenamento e processamento massivo de dados em sistemas distribuídos. O *Hadoop* oferece como ferramentas principais o *MapReduce*, responsável pelo processamento distribuído, e o *Hadoop Distributed File System* (HDFS), para armazenamento de grandes conjuntos de dados, também de forma distribuída. Embora seja um conceito relativamente recente, o *Apache Hadoop* tem sido considerado uma ferramenta eficaz, sendo utilizado por grandes companhias mundiais tais como IBM,

[1] https://hadoop.apache.org/

Oracle, Facebook, Yahoo! entre outras.

A Figura 5 ilustra o modelo arquitetural do *Apache Hadoop*, constituído pela HDFS, responsável pelo armazenamento distribuído e pelo *MapReduce*, que se encarrega da distribuição do processamento [3].

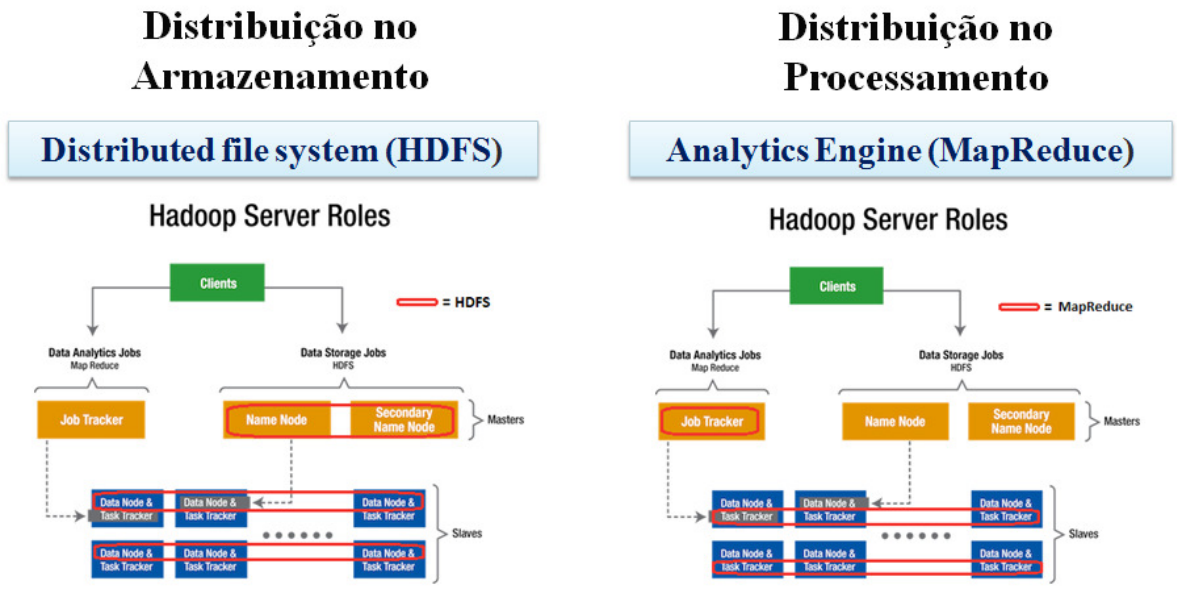


Figura 5 - Modelo arquitetural do *Apache Hadoop*.

O *framework Hadoop* é usado no processamento distribuído de grandes *datasets* (*tera* or *peta bytes* de dados) através de vários nós em *cluster* (centenas ou milhares de nós), oferecendo as seguintes características.

- Escalabidade (*petabytes* de data, centenas de máquinas).
- Flexibilidade nos formatos/ esquemas de dados (sem esquemas, desestruturados, social, logs, etc.).
- Mecanismos de tolerância a falhas simples e eficientes.
- *Hardware* pode ser adicionado em regime de *commodities*.
- O *Hadoop* juntamente com o *MapReduce* forma uma plataforma que permite fácil construção e execução de aplicações que processam grandes quantidades de dados.
- Seu processamento é baseado no motor *MapReduce*, um modelo que utiliza-se de uma técnica de “dividir para conquistar”.
- Suporta fortemente aplicações que usam um modelo de acesso *write-once-read-*

many.

- É naturalmente paralelizável através de um grande conjunto de computadores.
- Aproveita-se do alto *throughput* oferecido pelo HDFS.

3.3 HDFS - Sistema de Arquivos Distribuídos *Hadoop*

O *Hadoop Distributed File System* (HDFS) é um sistema de arquivos distribuídos que tem muitas semelhanças com os sistemas de arquivos distribuídos de alto custo existentes. No entanto, aquilo que o difere dos demais é alta tolerância às falhas pelo HDFS, o qual foi projetado para ser implementado em *hardware* de baixo custo e possui alta escalabilidade. O HDFS fornece acesso a informação com alta taxa de transferência de dados para aplicativos e é adequado para aplicações com grandes conjuntos de dados.

O HDFS foi originalmente construído como infraestrutura para o projeto do motor de busca da *web Apache Nutch*, e é um subprojeto da plataforma *Apache Hadoop*. A plataforma de desenvolvimento *Apache Hadoop* foi desenhada para resolver problemas onde há uma grande quantidade de dados – talvez uma mistura de dados complexos e não estruturados. É recomendado em situações nas quais se deseja fazer uma análise profunda e computacionalmente extensa, como *clustering* e *targeting*.

O *Hadoop* foi desenvolvido pela *Apache Sofware Foundation* para ser executado em um grande número de máquinas que não compartem memória nem discos (conceito denominado *shared-nothing*). Quando os dados são carregados usando o *framework*, são divididos em blocos que depois se propagam através de diferentes servidores (Figura 6).

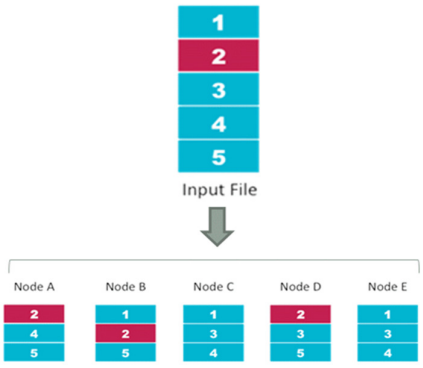


Figura 6 - Funcionamento do HDFS.

Além disso, devido ao grande número de cópias dos arquivos físicos entre diversos servidores, os dados armazenados em um servidor, que porventura, se desconecta ou morre, podem ser reproduzidos de forma automática a partir de uma cópia conhecida.

Hadoop também complementa os sistemas de bancos de dados existentes de quase qualquer tipo. *Hadoop* é excelente em armazenamento de dados de formatos, semi-, ou mesmo não estruturados, uma vez que permite que você decida como interpretar os dados em tempo de análise, o que lhe permite mudar a maneira de classificar os dados a qualquer momento: uma vez que você atualizou os algoritmos, basta executar a análise novamente.

Em um sistema de banco de dados centralizado, um disco com grande capacidade de armazenamento é conectado a vários processadores (de 4 a 16 processadores, por exemplo). Mas esta é toda a potência que pode chegar. Em um *cluster Hadoop*, cada um destes servidores possui dois, quatro ou oito CPUs. É possível executar o trabalho de indexação mediante o envio de seu código a cada uma das dezenas de servidores no *cluster* e cada servidor funciona na sua própria porção de dados. Os resultados se entregam de novo a um todo unificado. Isto é *MapReduce*: atribuir a operação a todos os servidores e depois reduzir os resultados de novo em um único conjunto de resultados.

Arquitetonicamente, a razão pela qual é capaz de fazer frente a uma grande quantidade de dados, é que faz o *Hadoop* ser altamente escalável. E essa é a razão pela qual o Hadoop é capaz de fazer perguntas computacionalmente complicadas: porque tem todos vários processadores (no caso computadores em rede) que trabalham em paralelo. O mecanismo responsável por essa computação em paralelo é o *MapReduce*, que será visto a seguir.

3.4 *MapReduce*

O *Map/Reduce* é um *framework* computacional para processamento paralelo criado pela Google. Ele é um motor de processamento que abstrai as dificuldades do trabalho com dados distribuídos, de modo que cada nó é independente e autossuficiente, eliminando quaisquer problemas que o compartilhamento de informações possa trazer (hadoop.apache.org). A Figura 7 ilustra as três funções do MapReduce [3].

- *Map*: recebe uma lista como entrada e aplica uma função para gerar uma nova lista como saída. Particularmente no uso em conjunto do *Hadoop* com técnicas de *Map/Reduce*, as funções *Map* utilizam os blocos do arquivo armazenado como entrada. Podem ser, por exemplo, uma linha em um arquivo de *log* ou de uma tabela.

Os blocos podem ser processados em paralelo em diversas máquinas do aglomerado. Como saída, as funções *Map* produzem, normalmente, outros pares chave/valor.

- *Shuffle*: a etapa de *shuffle* é responsável por organizar o retorno da função *Map*, atribuindo para a entrada de cada *Reduce* todos os valores associados a uma mesma chave. Esta etapa é realizada pela biblioteca do *MapReduce*.
- *Reduce*: recebe o resultado da função *Map* como entrada, aplica uma função para que a entrada seja reduzida a um único valor na saída. No *Hadoop* as funções *Reduce* são responsáveis por fornecer o resultado final da execução de uma aplicação, juntando os resultados produzidos por funções *Map*, ou seja, os conjuntos de valores associados a uma chave são agrupados em lista e consumidos, retornando uma nova lista de pares chaves/valor.

Um trabalho (*job*) pode ser considerado como uma execução completa de um programa *Map/Reduce* incluindo todas as suas fases: *Map*, *Shuffle* e *Reduce*. Uma tarefa (*task*) é definida como a execução de uma função (*Map* ou *Reduce*) dentro do *framework*.

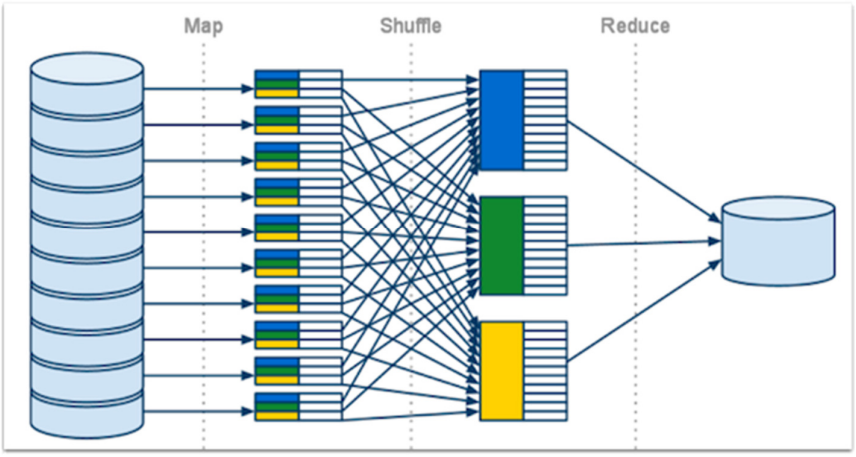


Figura 7 - Funcionamento do MapReduce.

3.5 Ecossistema *Hadoop*

A grande quantidade de dados gerada nos ambientes computacionais atuais trouxe a necessidade de se criar alternativas capazes de promover um processamento mais rápido e eficaz que os fornecidos pelos bancos de dados relacionais. Por ser um projeto de código

aberto, o *Apache Hadoop* tem recebido excelentes contribuições no seu desenvolvimento. Ao passo que novas necessidades vêm à tona, novas ferramentas estão sendo criadas anualmente para serem executadas sobre o *Hadoop*. A Figura 8 mostra os principais subprojetos do ecossistema *Hadoop*.

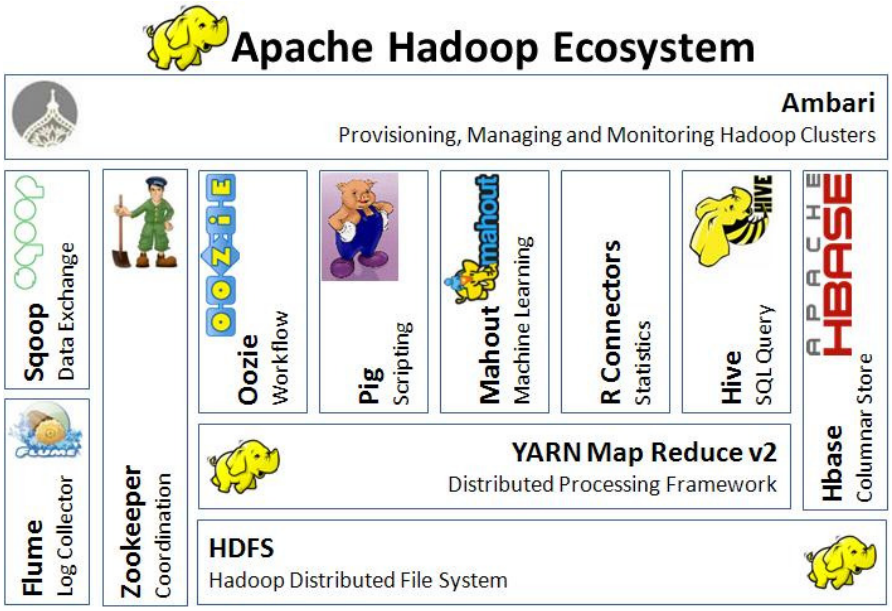


Figura 8 – Subprojetos do ecossistema Hadoop.

O *Hadoop* tem sido uma área amplamente explorada, especialmente pela flexibilidade que oferece aos desenvolvedores, podendo ser utilizado como infraestrutura nas mais diversas áreas da ciência, e também pela facilidade de uso e robustez proporcionada pelo projeto como um todo.

4 DIMENSÕES ENVOLVIDAS NO CONCEITO DE NoSQL

4.1 Consistência Eventual ou Relaxada

Como os conflitos de atualização são eventualmente processados durante a fase de armazenamento, a consistência está em garantir que um banco de dados esteja sempre íntegro aos seus clientes. Se conflitos de gravação e escrita ocorrem quando dois clientes tentam escrever os mesmos dados ao mesmo tempo (concorrência), a consistência é observada quando um cliente lê dados inconsistentes no meio da gravação de outro cliente. Por isso, cada operação do banco de dados deve levar o seu estado de um estado consistente para outro estado consistente [4].

O teorema CAP afirma que se você estabelece dados particionados em rede (tolerância a partição) e ainda oferece garantia de disponibilidade de dados, a consistência tem que ser relaxada. Em sistemas de armazenamento distribuídos que estabelecem cópias dos blocos de dados (como o *Hadoop*), esse conflito pode ainda se dar pois alguns nós podem ter recebido atualizações enquanto outros nós não.

Porém, os clientes geralmente querem consistência de leitura e escrita, o que significa que um cliente pode escrever e logo em seguida ler esse novo valor. Isto pode ser difícil se a leitura e a escrita acontecer em nós diferentes. Para obter uma boa consistência, é necessário envolver muitos nós em operações de dados, mas isso aumenta a latência.

A consistência eventual significa que em algum momento o sistema vai se tornar consistente entre essas cópias, uma vez que todas as gravações tenham se propagado para todos os nós. Assim, muitas vezes, você tem que abandonar essas consistências para garantir ou negociar uma boa latência. Em geral, a durabilidade também deve ser negociada, especialmente se você quiser ser tolerante às falhas com os dados replicados. Isso se dá para atingir a consistência forte, pois os modelos precisam comprometer a escalabilidade e o desempenho. Em outras palavras, os dados precisam ser bloqueados durante o período de atualização ou de processo de replicação para garantir que nenhum outro processo atualize os mesmos dados.

A consistência pode ser classificada pela ordem de suas propriedades ou pela diminuição das garantias oferecidas aos consumidores. Os modelos de consistência são os seguintes. [4]

a) Modelos de Consistência Forte.

- Consistência Rigorosa: as alterações nos dados são atômicas e têm efeito instantaneamente. Esta é a forma mais elevada de consistência.
- Consistência Sequencial: cada cliente vê todas as mudanças na mesma ordem em que foram aplicadas.

b) Modelos de Consistência Fraca.

- Consistência Eventual: quando as atualizações ocorrem por um período de tempo, no qual eventualmente todas as atualizações serão propagadas, e assim todas as réplicas estarão consistentes.

Ainda sobre a consistência, alguns modelos modernos de armazenadores NoSQL preveem o uso de um *cache* avançado. *Memcached/Memtable* são sistemas de *cache* de memória distribuída de propósito geral que é frequentemente usado para acelerar bases de dados através do *cache* de dados e de objetos na memória RAM, os quais permitem reduzir o acesso ao armazenamento de dados (Figura 9).

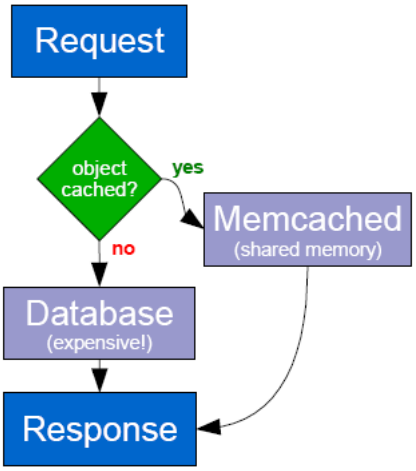


Figura 9 - *Memcached*

A existência dos dados (tabelas) no *cache* (*memtable*) é um modelo obscurecido de consistência eventual, pois o estado atual é uma interpretação dos dados em memória em conjunto do que está armazenado (Figura 10).

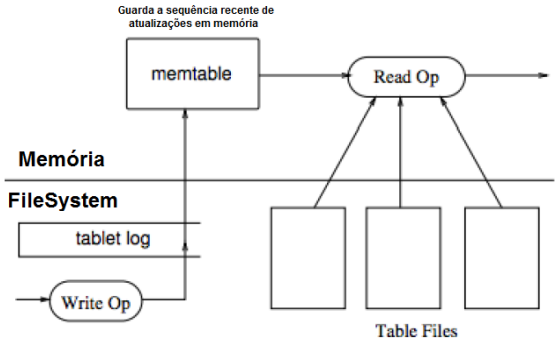


Figura 10 - *Google BigTable* e *Memcached/Memtable*

Cabe lembrar que o Sistema de Nome de Domínio - DNS (*Domain Name System*) da Internet é um exemplo bem conhecido de sistema com um modelo de consistência eventual. Os servidores DNS não necessariamente refletem os valores mais recentes, mas os valores são armazenados em *cache* e replicados em muitos diretórios na Internet. É necessário certo tempo para replicar os valores modificados em todos os servidores e clientes DNS. No entanto, o sistema DNS é extremamente bem-sucedido e se tornou uma das bases da Internet. Ele é altamente disponível e provou ser extremamente escalável, permitindo pesquisas de nome para mais cem milhões de servidores em toda a Internet.

4.2 Modelo de Dados Sem Esquema (Esquema Flexível)

Os novos bancos de dados possuem algumas características em comum, são armazenadores de pares de chaves e valores, ou seja, salvam, como o nome sugere, um conjunto de entradas formadas por uma chave associada a um valor e o valor pode ser de qualquer tipo, um binário ou *string* que está sendo salvo sem nenhum esquema pré-definido (*schema-free*). Diferentemente dos bancos SQL, na maioria dos armazenadores NoSQL não existe um esquema forte (diz-se que tem um esquema flexível). Essa abordagem facilita a distribuição dos dados entre vários servidores, nos quais cada servidor possui apenas uma fatia dos dados (*shard*) [5].

Assim, armazenadores livres de esquema ou com esquema flexível permitem que você adicione livremente campos para registros, não obstante existe um esquema implícito esperado pelos usuários dos dados. Frequentemente, mecanismos de *Map/Reduce* são usados para dar significado a agregação de dados, e assim, podem tratar a existência de padrões observáveis dentro do que está sendo armazenado.

4.3 Índices

Índices permitem classificar e acessar tabelas com base em diferentes campos e ordens de classificação. As opções aqui vão desde sistemas que não têm absolutamente nenhum índice secundário e sem ordem de classificação garantida (como um *HashMap*, ou seja, você precisa saber as chaves) até alguns que são fracamente apoiados por índices. Alguns modelos NoSQL tendem a simular a existência de índices pela formação de modelos secundários que apontam para as mesmas informações.

4.4 Compressão

Quando você tem que armazenar *terabytes* de dados, especialmente do tipo que consiste em texto legível, é vantajoso ser capaz de compactar os dados para obter economias substanciais em armazenamento bruto necessário. Alguns algoritmos de compressão podem atingir uma redução de até um décimo do espaço de armazenamento necessário. Alguns armazenadores NoSQL, implicitamente os Orientado a Colunas (tipologia a ser definida no decorrer deste documento), tendem a fornecer tais características em suas implementações, principalmente porque a entropia de diversas colunas apresenta conjunto de elementos idênticos, o que, como sabido, favorece a maiores taxas de compressão.

4.5 Escalabilidade e Disponibilidade pela Distribuição

Com o aumento da necessidade de dados e informações, o alto volume de acessos usando banco de dados distribuídos para processamento massivo (de dados) vem sendo utilizado. Neste sentido, bancos de dados paralelos e/ou distribuídos estão começando a substituir os modelos tradicionais, tanto no processamento de transações como em soluções de consultas tais como ambientes de BI.

Banco de Dados Distribuídos é um conjunto de múltiplos bancos de dados, logicamente inter-relacionados, distribuídos sob uma rede de computadores, o que o difere de bancos de dados centralizados e descentralizados (Figura 11). No conjunto de sistemas de suporte a banco de dados distribuídos, um SGBD distribuído deve prover todos os recursos usuais de um SGBD relacional, além de tornar transparente a distribuição ao usuário.

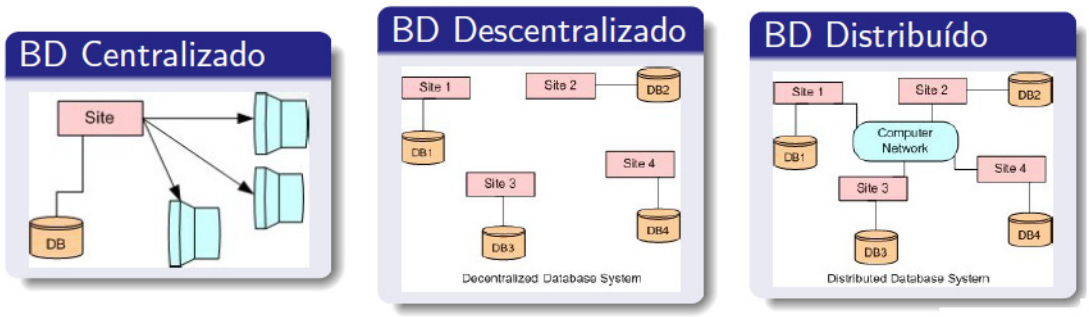


Figura 11 – Modelos de SGBDs segundo sua distribuição

O armazenamento de dados em diversos nós possui dois enfoques para os bancos de dados distribuídos (BDD), a saber.

a) Replicação de dados: a replicação de dados significa que um determinado objeto de dados lógico pode possuir diversos representantes armazenados em nós. O grau de suporte para a replicação é um pré-requisito para atingir o verdadeiro potencial de um sistema distribuído.

b) Partição (Fragmentação) de dados: uma relação é dividida em partições (fragmentos), na qual cada partição contém informação suficiente para permitir a reconstrução da relação original.

Esse enfoque ainda pode ser combinado - Fragmentação e Replicação de Dados – no qual a relação é particionada (em vários fragmentos) e o sistema mantém diversas réplicas de cada partição (fragmento). As técnicas de particionamento e replicação podem ser aplicadas sucessivamente a uma mesma relação. Uma partição pode ser replicada e as réplicas podem ser particionadas novamente (e assim por diante).

Em um BD distribuído, particionado ou replicado, devem ser tratados os seguintes aspectos.

- Independência de dados.
- Transparência de rede.
- Transparência de replicação.
- Transparência de fragmentação.

Existem duas formas de fazer a particionamento (Figura 12).

- Particionamento Horizontal (*Sharding*): divide a relação separando as tuplas em duas ou mais partições segundo alguma definição/campo. Assim cada nó

(servidor) atua como fonte única de um subconjunto de dados.

- Particionamento Vertical (Orientação a Colunas): divide a relação pela decomposição do esquema total existente em dois ou mais esquemas menores com as mesmas chaves. Assim diversos campos do esquema (flexível ou não) são distribuídos pelos diversos nós.

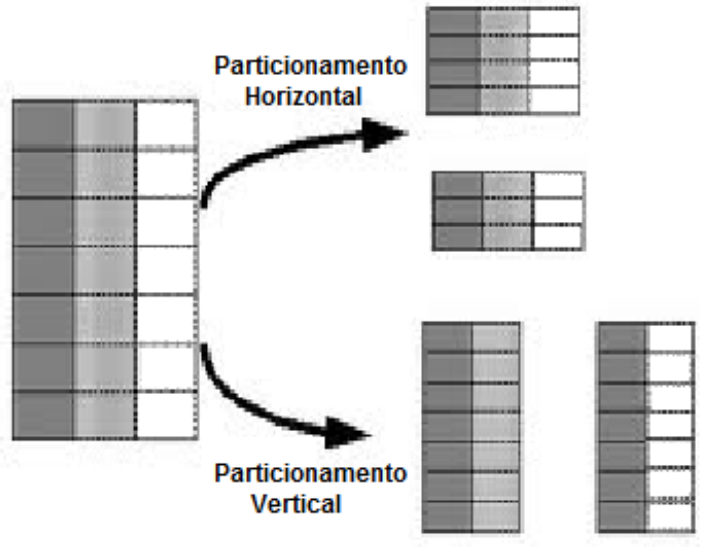


Figura 12 – Particionamento Horizontal (grupos de registros) e Vertical (colunas / campos)

4.5.1 Particionamento Horizontal: *Sharding*

O termo *sharding* descreve a separação lógica de registros em partições horizontais (Figura 13). A separação dos valores para essas partições é realizada em fronteiras fixas: você tem que estabelecer regras fixas para valores de rota para sua aplicação. Com isto, surge a dificuldade inerente de ter de reparticionar (*resharding*) os dados quando uma das partições horizontais excede a sua capacidade.

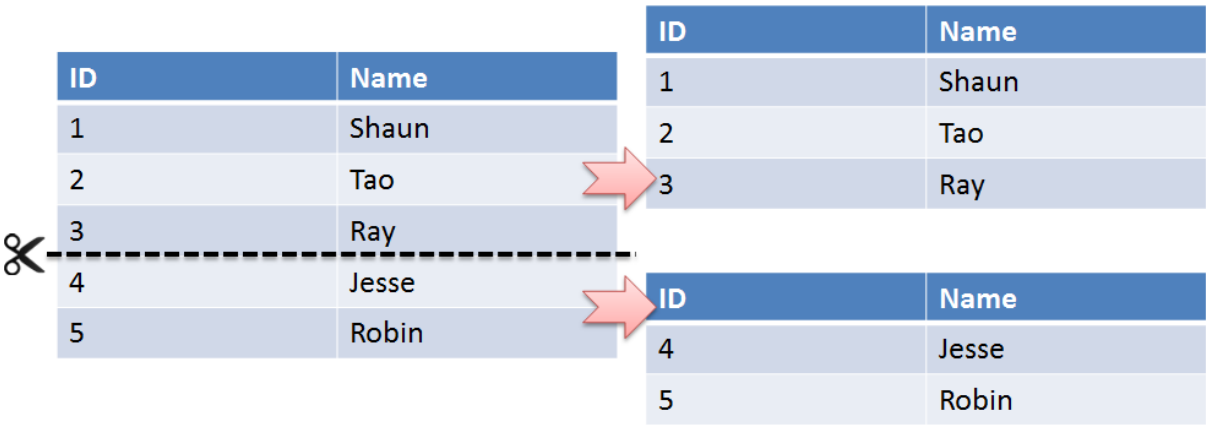


Figura 13 – *Sharding* - Particionamento Horizontal

O *resharding* é uma operação muito custosa, uma vez que o *layout* de armazenamento tem que ser reescrito. Isto implica a definição de novos limites e depois a divisão horizontal das linhas entre eles. Operações de cópia em massa podem acarretar em um esforço enorme sobre o desempenho de I/O, bem como os requisitos elevados de armazenamento temporariamente. E você ainda pode levar a impedimentos temporários nas atualizações oriundas de aplicativos clientes, precisando negociar as atualizações durante o processo *resharding*.

Isso pode ser atenuado pelo uso de fragmentos virtuais, que definem uma série de particionamentos de chaves em cada servidor atribuindo um número igual de fragmentos. Quando são adicionados novos servidores, apenas fragmentos podem ser transferidos para eles. Porém, isso ainda requer que os dados sejam movidos para o novo servidor.

4.5.2 Particionamento Vertical: Armazenamento em Colunas

Os modelos de armazenamento em linhas dos SGBDs tradicionais levam em consideração que as operações de BD mais onerosas são as que envolvem as pesquisas em discos rígidos, pois os esses são organizados numa série de blocos de tamanho fixo, tipicamente suficiente para armazenar várias linhas da tabela.

A fim de melhorar o desempenho geral, dados devem ser armazenados de forma a minimizar o número de pesquisas. Através da organização dos dados em tuplas de linhas relacionadas, as quais são agrupadas em conjuntos, o número de blocos que precisam ser lidos ou pesquisados é minimizado.

Assim, na comparação entre estruturas de dados orientada a coluna e orientados a linhas existe uma forte preocupação com a eficiência de acesso do disco rígido para uma

determinada carga de trabalho, pois o tempo de busca é extremamente longo quando comparado com os outros atrasos em computadores. Às vezes, a leitura de um *megabyte* de dados sequencialmente armazenados é considerada um acesso aleatório. Além disso, como o tempo de busca (I/O) é incrementado muito mais lentamente do que o crescimento do poder das CPU (Lei de *Moore*), e este foco provavelmente continuará em sistemas que dependem de discos rígidos para armazenamento, com exceção dos bancos de dados em memória (*in memory*). Assim:

a) a organização de dados orientada a colunas é mais eficiente quando uma leitura precisa ser feita ao longo de muitas linhas, mas apenas para um subconjunto notavelmente menor, porque a leitura de um menor subconjunto de dados pode ser mais rápida do que ler todos os dados;

b) a organização de dados orientada a colunas é mais eficiente quando os novos valores delas são fornecidos para todas as linhas de uma só vez, porque os dados de cada coluna podem ser escritos de forma mais eficiente, e, assim, substituí-los da coluna sem afetar quaisquer outras colunas que estão enfileiradas;

c) a organização de dados orientada a linhas é mais eficiente quando muitas colunas de uma única fileira são necessárias ao mesmo tempo, e quando o tamanho da linha é relativamente pequeno, como no caso de uma linha inteira que pode ser obtida com uma única pesquisa em disco;

d) a organização de dados orientada a linhas é mais eficiente quando se escreve uma nova linha e quando todos os dados são fornecidos ao mesmo tempo, e, assim, toda a linha pode ser reescrita com um único acesso a disco.

Na prática, estruturas de armazenamento orientadas a linhas são adequadas para cargas de trabalho de OLTP que são mais fortemente carregadas de transações interativas. Estrutura de armazenamento orientadas a colunas são mais bem adaptadas para cargas de trabalho de semelhantes a OLAP (por exemplo, *data warehouses*), que normalmente envolvem um menor número de consultas de alta complexidade ao longo de todos os dados (possivelmente *terabytes*).

Não obstante, as vantagens apresentadas sobre o particionamento horizontal e seu devido tratamento através de técnicas de *Map/Reduce* devem ser consideradas com cautela, porque a análise de dados de várias partições requer consultas que unem

diferentes armazenadores. Adicionalmente, se diversas colunas distribuídas em vários nós forem armazenadas instantaneamente, o modelo de particionamento horizontal deve prever uma consistência de atualização baseada no tempo (utilizando técnicas de *Vector Clocke*/ou *State Transfer Model*).

4.6 Consistência Histórica (Versionamento de Dados)

Em diversos modelos de armazenadores NoSQL, marcadores de *timestamp* são usados para suportar o versionamento, o que ajuda a detectar conflitos. Quando ocorre uma leitura e posteriormente uma atualização, o usuário pode conferir o registro de versão para assegurar que ninguém atualizou os dados entre sua operação de leitura e escrita. Os marcadores de *timestamp* podem ser implementados usando contadores, GUIDs, *hashes* de conteúdo, data e hora, ou uma combinação destes. Assim, em sistemas distribuídos, um vetor de marcadores de *timestamp* permite detectar quando diferentes servidores possuem atualizações conflitantes.

4.7 Persistência Poliglota

De maneira abstrata, o resultado mais importante da ascensão do NoSQL é a persistência poliglota. A persistência poliglota trata do uso de diferentes tecnologias de armazenamento de dados para lidar com diferentes necessidades de armazenamento. O uso da persistência poliglota é facilitado pelo encapsulamentodo de acesso a dados para os serviços, o que reduz o impacto das escolhas de armazenamento de dados em outras partes do sistema.

Porém, a adição de várias tecnologias de armazenamento de dados, obviamente, aumenta a complexidade do sistema tanto em programação quanto em operações, de modo que deve-se encontrar um equilíbrio entre as vantagens e a complexidade do uso de modelos diferenciados de armazenamento de dados.

5 ARMAZENADORES NOSQL: TIPOLOGIA NOTÓRIA

A principal motivação para o desenvolvimento de armazenadores NoSQL foi para dar suporte a implementação da *Web* 2.0, que aumentou o uso e a quantidade de dados armazenados em bases de dados. Além de dar suporte ao armazenamento de grandes volumes de dados, os armazenadores NoSQL são projetados para lidar com vários tipos de falhas.

O divisor de águas no movimento NoSQL foi a publicação de 2 artigos, a saber.

a) *BigTable*: *A Distributed Storage System for Structured Data*. Publicado pelo Google em novembro de 2006 no 17º Simpósio em Design e Implementação de Sistemas Operacionais [6].

b) *Dynamo*: *Amazon's Highly Available Key-Value Store*. Publicado pela Amazon em outubro de 2007 no 12 º Simpósio em Princípios de Sistemas Operacionais [7].

Os armazenadores NoSQL não são projetados para abandonar SQL ou mesmo a linguagem de consulta usada para recuperar informações de bancos de dados relacionais tradicionais, tais como Oracle e MySQL. Um nome melhor seria defini-los como soluções de "banco de dados não-relacionais", pois não usam o modelo de tabelas normalizadas de dados que sustentam as bases de dados relacionais.

Seu surgimento se deu dada uma crescente necessidade de disponibilização dos serviços *on-line* da Google e da Amazon, os quais necessitavam de novas formas de armazenar grandes quantidades de dados através de um número cada vez maior de servidores, de modo que cada um criou uma nova plataforma de *software* para fazê-lo (a Google construiu o *Google BigTable* e a Amazon construiu o *Dynamo*).

O resultado foi um número muito grande de bancos de dados NoSQL projetados especificamente para funcionar através de milhares de servidores. Estas plataformas de s*oftware* da nova era incluem o *Cassandra*, *HBase*, e *Riak*, os quais ajudam outros gigantes da *web*, incluindo Facebook e Twitter, assim como empresas mais tradicionais a processarem uma grande massa de dados.

Os modelos orientados a documentos foram inspirados pelo *Lotus Notes*, uma plataforma de colaboração *on-line* desenvolvida originalmente na década de 1970 e 80. Os bancos de dados que estão sob estes rótulos não exigem esquemas de tabela fixa e, geralmente, não suportam instruções e operações de junção SQL. Tendências em arquiteturas de computadores, como a computação na nuvem, e a necessidade crescente

de prover serviços escaláveis estão pressionando os bancos de dados numa direção para a qual eles necessitam oferecer escalabilidade horizontal. É importante entender que o intuito não é eliminar bancos de dados relacionais, mas oferecer uma alternativa.

No Quadro 2 apresentamos um histórico das soluções existentes, para suportar a discussão do processo evolutivo para solução de problemas notórios no armazenamento de dados.

Quadro 2 – Histórico de soluções em armazenamento de dados

Year	*System/ Paper*	*Scale to 1000s*	*Primary Index*	*Secondary Indexes*	*Transactions*	*Joins/ Analytics*	*Integrity Constraints*	*Views*	*Language/ Algebra*	*Data model*	*my label*
1971	RDBMS	O	✔	✔	✔	✔	✔	✔	✔	tables	sql-like
2003	memcached	✔	✔	O	O	O	O	O	O	key-val	nosql
2004	MapReduce	✔	O	O	O	✔	O	O	O	key-val	batch
2005	CouchDB	✔	✔	✔	record	MR	O	✔	O	document	nosql
2006	BigTable (Hbase)	✔	✔	✔	record	compat. w/MR	/	O	O	ext. record	nosql
2007	MongoDB	✔	✔	✔	EC, record	O	O	O	O	document	nosql
2007	Dynamo	✔	✔	O	O	O	O	O	O	key-val	nosql
2008	Pig	✔	O	O	O	✔	/	O	✔	tables	sql-like
2008	HIVE	✔	O	O	O	✔	✔	O	✔	tables	sql-like
2008	Cassandra	✔	✔	✔	EC, record	O	✔	✔	O	key-val	nosql
2009	Voldemort	✔	✔	O	EC, record	O	O	O	O	key-val	nosql
2009	Riak	✔	✔	✔	EC, record	MR	O			key-val	nosql
2010	Dremel	✔	O	O	O	/	✔	O	✔	tables	sql-like
2011	Megastore	✔	✔	✔	entity groups	O	/	O	/	tables	nosql
2011	Tenzing	✔	O	O	O	O	✔	✔	✔	tables	sql-like
2011	Spark/Shark	✔	O	O	O	✔	✔	O	✔	tables	sql-like
2012	Spanner	✔	✔	✔	✔	?	✔	✔	✔	tables	sql-like
2012	Accumulo	✔	✔	✔	record	compat. w/MR	/	O	O	ext. record	nosql
2013	Impala	✔	O	O	O	✔	✔	O	✔	tables	sql-like

Rick Cattel's clustering from
"Scalable SQL and NoSQL Data Stores"
SIGMOD Record, 2010

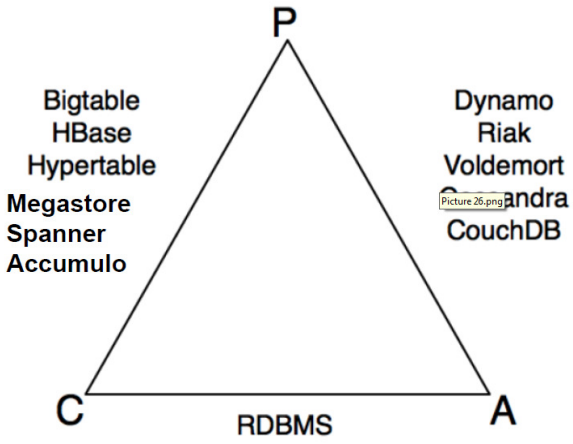
extensible record stores

document stores

key-value stores

Legenda: *check* (atende), círculo (não atende), / (atende parcialmente)

A Figura 14 ilustra a relação entre os mais notórios armazenadores NoSQL modernos perante o paradigma CAP, além de enquadrar a posição dos SGBDs relacionais.



src: Shashank Tiwari

Figura 14 – Notórios Armazenadores NoSQL e SGBDs relacionais (RDBMS) e o CAP.

5.1 Armazenamento Chave-Valor (*Key-Value Stores*)

a) Conceituação

A ideia principal desse armazenador é usar uma tabela *hash*, na qual há uma chave única e um indicador de um dado ou de um item em particular. O modelo chave-valor é o mais simples e fácil de implementar, mas ele é ineficiente quando você somente está interessado em consultar ou em atualizar parte de um valor, entre outras desvantagens. É um modelo simples que permite a visualização do banco de dados como uma grande tabela *hash*.

b) Exemplos.

- *Dynamo DB* (Amazon).
- *Voldemort.*
- *Oracle Berkeley DB.*
- *Scalaris.*
- *Redis.*
- *Tokyo Cabinet/Tyrant*.

c) Aplicações típicas.

Cache de conteúdo (foco em escalar para imensas quantidades de dados).

d) Modelo de dados.

Coleção de pares de chave-valor.

e) Pontos fortes.

Pesquisas rápidas.

f) Fraquezas.

Dados armazenados não têm *schema*.

5.2 Armazenamento Orientado a Colunas (*BigTable-style Databases*)

a) Conceituação.

Foram criados para armazenar e para processar grandes quantidades de dados distribuídos em muitas máquinas. Ainda existem chaves, mas elas apontam para colunas múltiplas. Essas são organizadas por família da coluna. É um estilo emergente de banco de dados que teve como referência o modelo *BigTable* do Google e que dentre suas características podemos destacar o particionamento vertical dos dados com forte consistência dos mesmos.

b) Exemplos.

- *Apache HBase.*
- *Cassandra.*
- *Google BigTable.*
- *Hyperbase.*
- *Hypertable.*
- *Apache Accumulo.*
- HPCC.
- *Splice Machine.*

c) Aplicações típicas.

Sistemas de arquivos distribuídos.

d) Modelo de dados.

Colunas e famílias de colunas.

e) Pontos fortes.

Pesquisas rápidas.

Boa distribuição de armazenamento de dados.

f) Fraquezas.

API de baixo nível.

5.3 Armazenamento Orientado a Documentos (*Document Databases*)

a) Conceituação

Foram inspirados por *Lotus Notes* e são similares ao armazenamento chave-valor. O modelo consiste basicamente de documentos versionados que são coleções de outras coleções de chave-valor. Os documentos semiestruturados são armazenados em formato JSON. Bancos de dados de documentos são essencialmente o próximo nível do chave-valor, permitindo valores aninhados associados a cada chave. Bancos de dados de documentos suportam consultas mais eficientemente, pois armazenam coleções de documentos, ou seja, objetos com identificadores únicos e um conjunto de campos (*strings*, listas ou documentos aninhados), assemelhando-se ao modelo chave-valor, porém cada documento tem um conjunto de campos-chaves e valores destes campos.

b) Exemplos.

- *CouchDB*.

- *MongoDB*.

c) Aplicações típicas.

Aplicações *Web* (Similar ao armazenamento chave-valor, mas o BD sabe qual é o valor).

d) Modelo de dados.

Coleções de chave-valor.

e) Pontos fortes.

Tolerante a dados incompletos.

f) Fraquezas.

Desempenho na pesquisa.

Não apresenta uma sintaxe de pesquisa padrão.

5.4 Armazenamento Orientado a Grafos (*Graph Databases*)

a) Conceituação

Neste modelo, em vez de tabelas com linhas e colunas e a rígida estrutura do SQL, um modelo gráfico flexível é usado podendo, mais uma vez, ser escalado através de várias máquinas. Possui três componentes básicos (nós, relacionamentos, propriedades) que permitem que o SGBD possa ser visto como

um multigrafo rotulado e direcionado, onde cada par de nós pode ser conectado por mais de uma aresta.

b) Exemplos:

- *Neo4j.*
- *InfoGrid.*
- *Sones.*
- *HyperGraphDB*.

c) Aplicações típicas.

Redes Sociais, Recomendações (Foco em modelar a estrutura dos dados – interconectividade).

d) Modelo de dados.

Grafo de Propriedades – Nós.

e) Pontos fortes.

Suporte direto a algoritmos gráficos tais como caminho mais curto, conectividade, relacionamentos em n graus, etc.

f) Fraquezas.

Tem que atravessar todo o gráfico para obter uma resposta definitiva.

Não facilita solução de problemas de agrupamento / agregação.

6 CHAVE-VALOR (*KEY VALUE*)

6.1 Introdução

O modelo de armazenamento chave-valor é a forma mais simples de se associar um grande arquivo de dados com uma simples sequência de texto (valor chave). A utilização do modelo de armazenamento surgiu, dentre outras motivações, como uma alternativa para resolução de casos simples, nos quais as consultas dinâmicas por complexos conjuntos de dados e a grande quantidade de normalização que leva a hierarquias complexas das tabelas dos bancos relacionais vão no sentido contrário a simples ideia de armazenar uma chave única e um valor associado [7].

Tipicamente, implementações da arquitetura chave-valor não têm uma linguagem de consulta, a exemplo do SQL, pois a manipulação dos dados é executada através de funções para adicionar (*put*), recuperar (*get*) ou remover (*delete*) pares de chave-valor no banco de dados. Operações como as de busca, comparação ou ordenação com os valores armazenados na base de dados, entretanto, isto resulta em uma arquitetura totalmente "*schema-free*". Em complemento as funções "*get*", "*put*" e "*delete*", o modelo chave-valor como regra básica o fato de que as chaves armazenadas são únicas, ou seja, nunca poderão existir duas chaves com o mesmo valor.

Este modelo de armazenamento de dados é comumente associado a um dicionário, que possui uma lista de palavras, as chaves, e onde cada uma destas palavras possui uma ou mais definições, os valores. Assim como as palavras nos dicionários, as chaves armazenadas nesta estrutura de dados são ordenadas, de modo que não é necessário vasculhar toda a base de dados para localizar uma determinada entrada, o que torna a resposta rápida independentemente do número de itens armazenados e faz com que implementações deste modelo sejam facilmente escaláveis.

Um dos pontos positivos do armazenamento chave-valor é o fato de que o modelo prega que não se especifique tipos de dados para a "chave" e o "valor". Como estes elementos são tratados como campos do tipo BLOB, sem qualquer tipo de verificação, existe grande flexibilidade dos tipos de informação que podem ser armazenadas, como imagens, endereços de páginas *web*, chamadas de serviço *Web*, etc.

Os principais benefícios de se utilizar o modelo de armazenamento chave-valor são os seguintes.

1. Quando você tem uma interface simples do serviço de dados que é utilizada em todas as aplicações, seu foco se move do desenho arquitetural para coisas como criar níveis de serviço precisos para o seu serviço de dados. Em complemento, a especificação dos níveis de serviço e a simplicidade da estrutura do chave-valor possibilitam um maior investimento em ferramentas de monitoramento do nível de serviço e sistemas de notificação automática.
2. Uma interface de dados simples resulta em sistemas que tem maior escalabilidade e confiabilidade.
3. A portabilidade de uma aplicação depende da complexidade da interface do banco de dados. A utilização de poucas interfaces padronizadas, como as operações de "*put*" e "*get*", permite uma rápida migração dos dados entre bases de dados a um baixo custo.

6.2 Quando Usar

- Sessão de Informação Armazenamento.
- Perfis de usuário, preferências.
- Carrinho de compras de dados.

6.3 Quando Não Usar

- Relação entre os dados.
- *Multioperation transation*.
- Consulta por dados.
- Operação por conjuntos.

7 ORIENTADO A COLUNAS (*COLUMN-STORES*)

7.1 Introdução

No armazenamento de dados, tal como em banco de dados, a unidade mais básica é uma coluna. Uma ou mais colunas formam uma linha que é gerenciada por uma chave da linha. Assim, todas as linhas são sempre classificadas lexicograficamente por suas chaves [6].

Um número de linhas, por sua vez, forma uma tabela, podendo haver várias delas. Essa é uma descrição razoável para um banco de dados típico, mas, no caso de armazenadores NoSQL orientado a colunas, além do armazenamento de colunas poder se dar por particionamento horizontal, cada coluna pode ter várias versões, com cada valor distinto contido em uma célula separada; e essa dimensão extra (de permitir que várias versões de cada célula) é fato incomum nos SGBDs relacionais.

Nos armazenadores NoSQL orientados a colunas, as linhas são compostas de colunas, e aqueles que, por sua vez, podem ser agrupadas em famílias de colunas, conforme ilustrado na Figura 15.

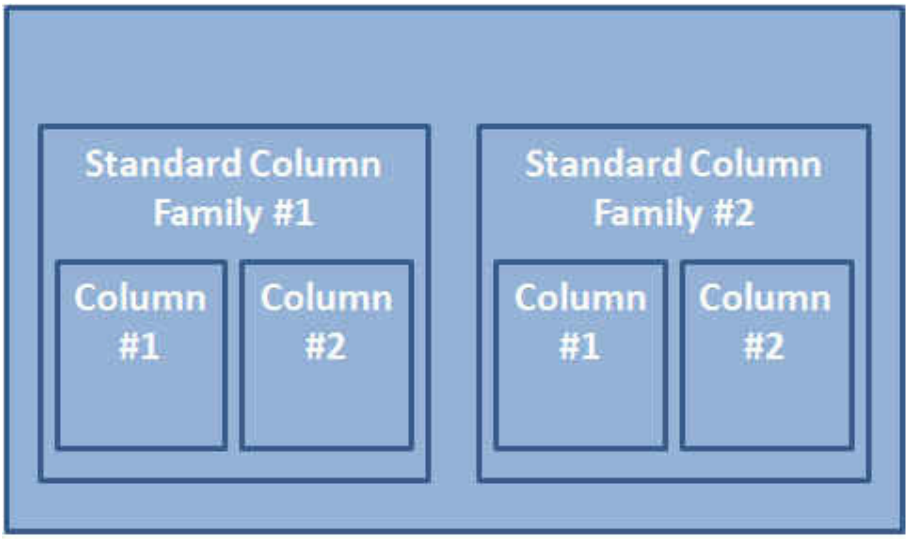


Figura 15 – Visão Conceitual de uma tabela, famílias de colunas e colunas.

Nesses armazenadores, todas as colunas em uma família de coluna são armazenadas juntos no mesmo arquivo de armazenamento de baixo nível, no caso do *Google BigTable*, chamada de *hFILE*. A formação de agrupamentos com níveis diferenciados nas colunas ajuda na construção limites semânticos ou tópicos entre os dados (árvores de dados) e

também na aplicação de determinadas características a elas, por exemplo, a compressão ou registro permanente na memória. Por fim, todas as linhas e colunas são definidas no contexto de uma tabela.

Famílias de colunas devem ser definidas quando a tabela é criada e não devem ser alteradas muitas vezes, nem deve haver muitas delas. Existem algumas falhas conhecidas na implementação atual que forçam que esse número seja limitado a dezenas, mas na prática muitas vezes é um número muito menor. O nome da família de coluna deve ser composto por caracteres imprimíveis.

No versionamento de uma coluna, o usuário pode especificar o número de versões de um valor deve ser mantido. Além disso, há suporte para exclusões de predicados que lhe permite manter, por exemplo, apenas os valores escritos na semana passada.

Dos modelos de armazenadores NoSQL orientados a colunas, o *Google Bigta*ble originou e estabeleceu o modelo de armazenamento em colunas, como também aquele implementado pelo *HBase*, no qual foi distribuído e cada tabela é indexada por uma chave de linha, por uma chave de família de colunas (caso exista), por uma chave de coluna, ou um marcador de *timestamp*. Esses elementos, os quais formam a chave de acesso em conjunto representam seus dados (valores) da seguinte maneira:

(Tabela, Chave de Linha, Família de Colunas, Coluna, *Timestamp*) ⟵ Valor

Uma motivação para o armazenamento de valores por coluna baseia-se no pressuposto de que, para questões específicas, nem todos os valores são necessários. Este é frequentemente o caso em bancos de dados analíticos, portanto eles são bons candidatos para este esquema de armazenamento diferente.

Assim, a redução de I/O é uma das principais razões para este novo *layout*, pois oferece vantagens adicionais, já que os valores de uma coluna são frequentemente muito semelhantes na natureza ou mesmo na variação entre as linhas lógicas. Elas são, muitas vezes, muito mais adequadas para a compressão do que os valores heterogêneos de uma estrutura de registro orientada a linha.

Recursos: Consistência, Transações, Consultas.

7.1.1 *Auto-Sharding*

As unidades básicas de escalabilidade e balanceamento de carga em modelos

orientados a colunas são chamadas, geralmente, de *"Region's"*. As *Regions* são faixas contíguas essencialmente de linhas armazenadas em conjunto. Elas são divididas de forma dinâmica pelo sistema quando elas se tornam muito grandes. Alternativamente, elas também podem ser fundidas para reduzir o seu número e os arquivos de armazenamento necessários.

As *Regions* do *HBase* são equivalentes a banco de dados com partições que utilizam *sharding*. Essas partições podem ser espalhadas por vários servidores físicos, distribuindo assim a carga e, portanto, fornecendo escalabilidade.

Inicialmente, há apenas uma *Region* para uma *Table*, porém, com a adição de novos dados, o sistema monitora esse processo para garantir que você não exceda o tamanho máximo configurado. Se você ultrapassar o limite, a região é dividida em duas pelo valor médio das chaves gerando *Regions* estatisticamente de mesmo tamanho.

Cada *Region* é servida por exatamente um *Region Server*, porém cada um desses servidores pode servir muitas *Regions* a qualquer momento. A estrutura de *Regions* de modelos orientados a colunas permite a recuperação rápida quando um servidor falhar, e um balanceamento de carga *"fine-grain"*, uma vez que *Regions* podem ser movidas entre servidores quando a carga do servidor, que atualmente atende a região, está alta ou se esse servidor ficar indisponível devido a uma falha.

7.1.2 Versionamento

Uma característica especial do *HBase* é a possibilidade de armazenar várias versões de cada célula (o valor de uma determinada coluna). Isto é conseguido através de marcações de *timestamp* para cada uma das versões, e seu armazenamento se dá em ordem decrescente. Assim, quando você coloca um valor em *HBase*, você tem a escolha de fornecer um *timestamp* ou omitir esse valor, o que é então preenchido pelo *RegionServer* quando a operação é realizada. Se não for especificado o tempo nas chamadas de API do cliente, o tempo de servidor irá prevalecer.

7.1.3 Compressão de Dados

Se o armazenamento de dados em colunas é de uma única coluna, geralmente esta apresenta um conjunto de dados mais uniforme, isto é, com mais baixa entropia. Portanto, no armazenamento de dados em colunas há maiores oportunidades para otimizações do espaço de armazenamento do que o armazenamento de dados orientado a linhas; principalmente pelo uso de esquemas de compressão modernos populares, tais como LZW

ou codificação do tilo RLE (*run-length encoding*), que fazem uso da similaridade de dados adjacentes para compressão. Embora as mesmas técnicas possam ser utilizadas em dados orientados em linha, suas implementações típicas resultarão em compressões menos eficazes.

Ainda, com o uso de índices de *bitmap*, essa organização pode melhorar fortemente a compressão. Para maximizar os benefícios de compressão da ordem lexicográfica no que diz respeito a codificação RLE, o melhor é usar colunas de baixa cardinalidade (entropia) como chaves primárias. Por exemplo, dada uma tabela com as colunas sexo, idade e nome, seria melhor classificar primeiro a coluna sexo (cardinalidade de dois), depois a idade (cardinalidade < 150), e por último nome.

Os processos de compressão em modelos baseados em coluna resultam em uma redução no espaço de disco, em função da eficiência de recuperação. Esta eficiência de todos os dados a partir de uma única linha ainda é melhor quando os dados estão situados em um único local e usando uma arquitetura orientada a linha. Além disso, quanto maior compressão por adjacência alcançada (em modelos em colunas), mais difícil é o acesso aleatório, pois talvez todos os dados precisem ser descompactados para serem lidos. Portanto, arquiteturas orientadas a coluna são, por vezes, enriquecidas por mecanismos adicionais destinados a minimizar a necessidade de acesso aos dados comprimidos

Determinados armazenadores orientados a colunas implementam mecanismos específicos para acelerar o processo de teste da presença de uma chave de *hash* específico no contexto de seus dados. A Google, especificamente, implementa os *"Bloom Filters"* que são mecanismos otimizados de pesquisa de existência de uma chave de *hash*. Este tipo de teste permite a resposta muito mais rápida, reduzindo o acesso a disco durante leituras.

7.2 Quando Usar

- Registro de Eventos.
- Sistema de Gerenciamento de Conteúdo, plataformas de *blogs.*
- Contadores.

7.3 Quando Não Usar

- Resposta diversos campos por consultas diretas (vs SGBDs SQL).

8 ORIENTADO A DOCUMENTOS (*DOCUMENT-STORES*)

8.1 Introdução: Documentos e Mensagens

Os documentos utilizados na arquitetura de armazenamento orientada a documentos são vagamente definidos e podem ser serializados em JSON, XML, YAML, BSON ou em formato binário como PDF, DOC, XLS. Os documentos são registrados com um número único de identificação, o que faz com que o modelo de armazenamento se assemelhe muito com a estrutura chave-valor, sendo a identificação feita pela chave e pelo valor do documento. Estes documentos podem encapsular outros documentos, fazendo assim com que todas as informações referentes a uma chave fiquem armazenadas em um bloco isolado eliminando a necessidade de relações estritas entre documentos que não estejam encapsulados, entretanto isto também provoca a necessidade de duplicação de dados em mais de um documento, portanto fica a cargo do desenvolvedor escolher qual modelo a seguir.

No caso de sistemas relacionais, havendo a necessidade de acrescentar uma nova coluna, seria necessário alterar toda a estrutura da tabela, criando campos em todas as linhas já existentes o que acarretaria a criação de valores possivelmente inúteis. Em bancos orientados a documentos, podem-se acrescentar novos atributos a um documento sem que a mudança afete os demais. Isso se deve aos bancos de dados orientados a documentos não possuírem esquemas pré-definidos de armazenamento e não possuírem correlações estritas entre documentos.

Em bancos de dados relacionais, os dados precisam ser conexos para normalizar o armazenamento; já no caso de bancos de dados orientados a documentos, os dados relacionais serão inseridos no próprio documento, fazendo com que o sistema atue de maneira mais rápida, pois efetua apenas uma consulta para busca dos dados, fato que não ocorre nos bancos relacionais, pois eles precisam efetuar consultas adicionais em tabelas relacionadas.

A arquitetura voltada para documentos possui uma particularidade devido a liberdade de esquemas e mutabilidade do sistema. Os *softwares* de armazenamento que utilizam esta arquitetura podem ter suas características centrais muito diferentes de um software a outro. Outra possibilidade é a alteração por API's diversos, o que faz com que esta arquitetura consiga se adaptar para suprir de diferentes maneiras o teorema CAP,

possibilitando a apresentação de diferentes características inerentes aos sistemas ACID ou aos sistemas BASE. Estas adaptações muitas vezes fazem com que o banco fique menos eficiente.

Para melhor demonstrar o ponto que foi explicado acima, abaixo encontra-se a lista das características dos principais bancos orientados a documentos do mercado atualmente [8]:

	MongoDB	***CouchDB***	***RavenDB***
Documentos			
Formato	BSON	JSON	JSON
Metadata	Não	System	*System + Custom*
Versionamento	Não	Sim	*Plug-in* incluso
Attachments	GridFS	Sim	Sim
Map/Reduce	*JavaScript + others*	*JavaScript*	LINQ
Bulk Load	*Monogoimport utility*	Sim	Sim
Adhoc Query	Sim	Não	Não
Armazenamento			
Particionamento	A partir da versão 1.6	Sim	Sim
Durabilidade	"*Single Server*" disponivel desde a versão 1.8	"*crash-only*" *design*	Escrita com *log* e *snapshot* para recuperação de problemas via ESE
Transações	Não	Não	Sim
Concurrency	*Update in-place*	MVCC (*Multi-version Concurrency Control*)	*Optimistic concurrency*
Consistência	*Strong Master / Eventual Slave*	*Strong Node / Eventual Cluster*	Eventual
Replicação	*Master-Slave*	*Peer-based*	*Included Plug-in*
Interface			
Protocolo de Interface	Protocolo customizado de TCP/IP	HTTP/REST	HTTP/REST
.NET API	Projetos de terceiros	Projetos de terceiros	Nativo
Outros			
Triggers	Não	Sim	Sim
Segurança	Básica	Básica	Básica utilizando *plug-in* incluso
Linguagem de programação em que foi escrito	C++	*Erlang*	C#

Quadro 3 – Características dos bancos de dados orientados a documentos

8.2 Quando Usar

A arquitetura orientada a documento deve ser utilizada para soluções em que seja necessário o armazenamento de dados não padronizados, para que a sua liberdade de esquema possa ser utilizada com intuito de simplificar os processos de armazenamento e leitura em aplicações que devem escalar rapidamente, visto que a arquitetura favorece a elasticidade. Esta arquitetura também favorece a implementação rápida, pois sua natureza sem esquemas pré-definidos permite que os dados sejam armazenados e alterados sem extensivo planejamento prévio.

8.3 Quando Não Usar

A arquitetura orientada a documentos não deve ser utilizada por equipes de desenvolvedores que não estejam com um planejamento estratégico bem definido, pois sua natureza sem esquemas pré-definidos pode causar confusões no processo de escrita e leitura dos dados (alguns bancos possuem ferramentas que efetuam a organização dos dados automaticamente). Também não deve ser utilizada em soluções que não necessitem de características especificas deste modelo arquitetural, uma vez que o desempenho desta arquitetura (velocidade e volume de dados) pode ser inferior quando comparada com as demais[8].

9 ORIENTADO A GRAFOS (*GRAPH-STORES*)

Bancos de dados orientados a grafos permitem modelar os dados como entidades e relacionamentos entre essas entidades. A ideia básica desse tipo de banco é representar os dados e/ou o esquema dos dados como grafos dirigidos, ou como estruturas que generalizem a noção de grafos. O modelo de grafos é aplicável em casos nos quais as informações sobre a interconectividade ou a topologia dos dados são mais importantes, ou tão importante quanto a informação propriamente dita [9].

9.1 Introdução: Propriedades dos Grafos

Os bancos orientados a grafos são formados por três componentes básicos: os nós (são os vértices do grafo), os relacionamentos (são as arestas) e as propriedades (ou atributos) dos nós e relacionamentos, conforme ilustra a Figura 16.

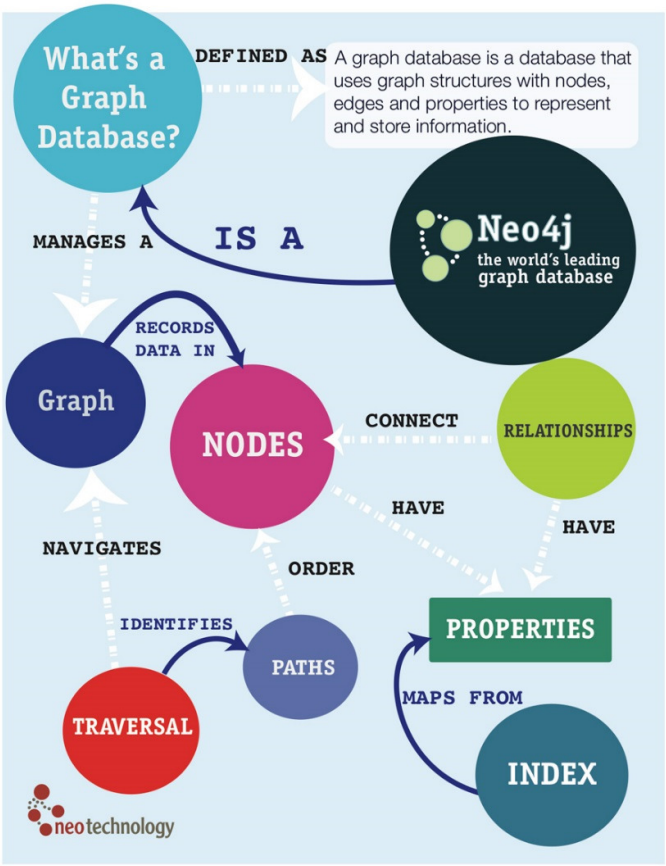


Figura 16 – Componentes básicos de um banco orientado a grafos.

A Figura 17 ilustra a representação de um banco orientado a grafos, exemplo utilizado como caso de uso do banco Neo4j, um dos mais conhecidos e utilizados.

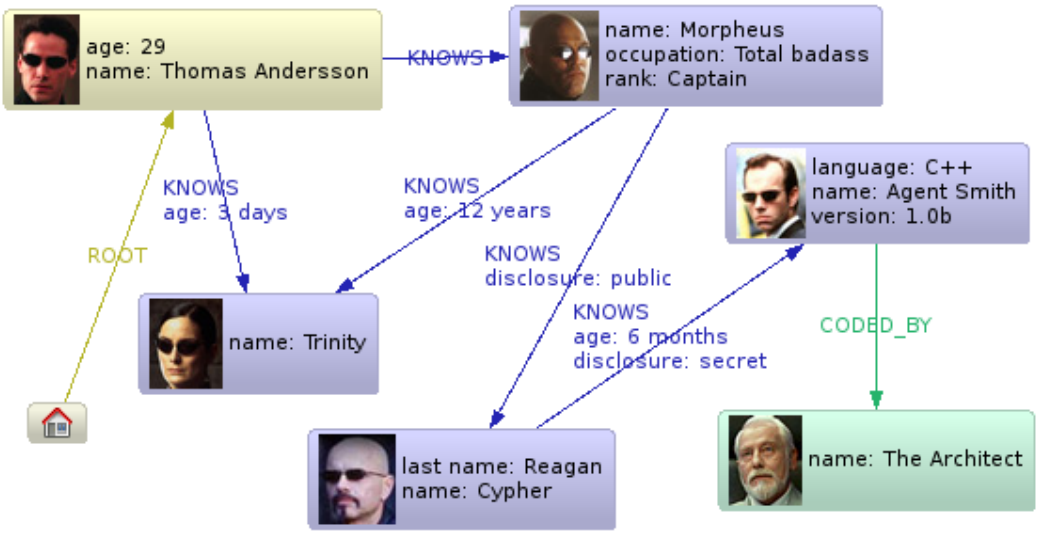


Figura 17 – Dados modelados através de grafos

Analisando o grafo, se quisermos saber quem o personagem Morpheu conhece, basta descobrirmos os relacionamentos de saída dele, assim como o tempo de amizade com os outros personagens.

Os bancos de dados relacionais tradicionais não possuem a característica de representação dos dados através de grafos de maneira natural, com isso, algumas pesquisas se tornam bastante complexas, ou até mesmo impraticáveis. Os bancos orientados a grafos permitem facilmente a localização de qualquer dado (característica conhecida como *traversing*), através dos nós e suas relações, sendo muito mais rápidos que uma *query* normal.

Esse é justamente o papel dos bancos orientados a grafos que permitem que dados sejam persistidos e percorridos, mantendo algumas características que se tornaram bastante comuns em bancos tradicionais, como controle de transações e seguindo as propriedades ACID, além de suporte a pesquisas textuais, através da integração com o *Apache Lucene* e o SOLR.

9.2 Quando usar

- Dados conectados (redes sociais): rede sociais são as que mais podem ser beneficiadas com o uso de banco de dados orientado a grafos. Não se trata apenas de identificar quem é amigo de quem em uma rede social. Um banco orientado a grafos também pode representar outras relações, conhecimentos, etc.

- Roteamento e serviços baseados em localização: relações entre entidades podem possuir como propriedade as métricas de distância e localização, podendo ser utilizadas para questões de roteamento e entrega de informação de forma mais efetiva. Por exemplo, pode-se fazer a recomendação de um determinado restaurante ou um ponto de interesse baseado na localização do usuário dentro do grafo.

- Mecanismos de recomendação: como os nós possuem relações entre eles dentro de um sistema, os bancos orientados a grafos são muito úteis para fazer recomendações baseadas nos relacionamentos existentes. Um fato interessante é que a medida que os nós e as relações aumentam de tamanho, o sistema de recomendação se torna ainda mais poderoso. Além disso, pode-se usar essas relações para a detecção de padrões de comportamento dentro do sistema.

9.3 Quando Não Usar

Em algumas situações, os bancos de dados orientados a grafos não são adequados. Quando se deseja atualizar a informação de todas as entidades ou um grande conjunto delas, os bancos orientados a grafos podem não ser adequados pelo fato de que realizar essa atualização em milhares de nós ou entidades, pode ser uma tarefa bastante complexa. Ou seja, os bancos orientados a grafos que possuem milhares de entidades e relações, podem ter seu desempenho afetado quando há a necessidade da realização de qualquer operação em toda a rede [9].

10 SEGURANÇA: CONFIDENCIALIDADE E PRIVACIDADE

10.1 Segurança no Armazenamento

Um modelo de garantir os princípios de confidencialidade em banco de dados baseia-se na capacidade de implementar modelos criptográficos junto ao armazenamento de dados. Para a consulta de dados criptografados é necessário um conjunto de criptossistemas que os trate adequadamente sem a perda de confidencialidade. Dessa forma, os dados brutos são cifrados de maneiras diferentes antes de serem inseridos em um banco de dados. Dependendo do tipo de consulta a ser realizada num determinado campo, uma técnica distinta será usada.

10.2 Probabilístico (*Random*)

A cifra é dita probabilística quando para valores diferentes existirão cifras diferentes com grande probabilidade. Para uma construção eficiente é usado uma cifra de bloco, como AES no modo CBC juntamente com um vetor de inicialização (IV) aleatório [10].

Provê máxima segurança devido a propriedade de indistinguibilidade sobre um Ataque de Texto em Claro Escolhido (*Chosen-plaintextAttack* – IND-CPA), um adversário será incapaz de distinguir pares de textos cifrados baseados em mensagens cifradas por ele. Apesar de garantir confidencialidade e integridade dos dados em um banco de dados, não é possível realizar nenhuma manipulação com os dados cifrados, exceto o comando *SELECT* básico com operações matemáticas de inteiros ou *floats* com o texto cifrado.

10.3 Pseudônimo (Determinístico)

O modelo de criptografia é dito determinístico quando é gerada a mesma cifra para a mesma mensagem em claro. O esquema é feito com a cifra de bloco AES no modo CBC com um IV de zeros. Assim, o esquema pseudônimo realiza uma permutação pseudoaleatória, o que representa uma diminuição no nível de segurança: adversários podem ter o conhecimento de quais valores cifrados correspondem ao mesmo valor em claro [11].

A escolha desse esquema permite a realização de consultas de igualdade, isto é, pode realizar o comando *SELECT* com predicados de igualdade, junções de igualdade, GROUP *BY*, *COUNT*, *DISTINT*, etc.

10.4 Busca por Palavras (*Searchwords*)

Este esquema não é necessariamente um esquema de criptografia. É calculada a

função *hash*, normalmente SHA-1, de todas as sequências possíveis de palavras. Em seguida, os *hashes* são mantidos em um campo e separados por espaços. Caso haja algum caractere especial inserido no texto, este deve ser informado no arquivo *schema*.

Representa o nível mais baixo de segurança implementado, uma vez que *hash* é uma função pública e facilmente reproduzida. Um ataque de dicionário conseguiria relacionar a mensagem em claro com o seu *hash* correspondente. Admite consultas do tipo *SELECT* com checagem de conteúdo (condição *WHERE* com atributo *CONTAINS*) com palavras-chaves inteiras.

10.5 Busca Probabilística por Palavras (*Probabilistic Searchwords*)

Oferece um nível maior de segurança que o modelo anterior, pois em vez de substituir os dados por uma sequência de *hashes* no campo, os dados são cifrados com o modelo probabilístico e um outro campo para permitir a busca sobre dados privados é acrescentado [12]. Primeiramente, o texto é dividido em palavras-chave, depois suas repetições são eliminadas e cada palavra é cifrada como estabelecido no referencial. Da mesma forma que a busca anterior, permite consultas *SELECT* com o uso de checagem de conteúdo.

10.6 Criptografia Homomórfica

Uma cifra C ( ) é dita homomórfica quando, dado C(x) e C(y), é possível obter C(x # y) sem descriptografar x e y, para alguma operação # [13]. A utilização de uma cifra puramente homomórfica é comprovadamente lenta, então a alternativa eficiente a ser escolhida é uma cifra parcialmente homomórfica, como Paillier, que é um esquema seguro IND-CPA. Para admitir a soma, a multiplicação de dois valores cifrados resulta na soma de seus valores, ou seja, HOMK(x). HOMK(y) = HOMK(x+y), com a multiplicação realizada em módulo.

Logo, devido às propriedades de uma cifra parcialmente homomórfica, são admitidas as consultas que buscam as manipulações numéricas de soma e média em colunas.

10.7 Criptografia de Preservação de Ordem

Tal modelo é equivalente a um mapeamento aleatório que preserva a ordem. Entretanto, devido a essa propriedade, torna-se mais fraca que um modelo determinístico, pois o adversário pode obter o conhecimento da ordem. O modelo permite relações de ordem entre dois dados baseadas nos seus valores cifrados, sem revelar os dados reais. Se uma coluna é criptografada com a preservação de ordem, o servidor pode realizar consultas inseridas em um determinado intervalo. Dessa forma, são permitidas consultas

de *ORDER BY*, *MIN*, *MAX*, *SORT*.

11 CONCLUSÃO

Por meio de um trabalho coordenado e interdependente entre as equipes do MJ/SE e da Universidade de Brasília, as atividades de elaboração deste RT foram planejadas, discutidas, executadas e documentadas.

Como modelo para avaliação e sugestões, este produto não representa um produto finalístico, sendo provável sua revisão após ciclos de análise e reavaliação e, principalmente, quando do atingimento de resultados parciais na realização das atividades envolvidas no subprojeto de Ecossistema do RIC.

Porém, o resultado de uma primeira avaliação deste documento pode permitir o início da construção de uma Prova de Conceito (PoC) mínima visando nortear a avaliação de quesitos específicos de infraestrutura, tais como: produtos e ferramentas que suportam os modelos e arquétipos sugeridos; questões de desempenho e curvas de ruptura na entrega de serviços; identificação de requisitos mínimos de *hardware* de infraestrutura de armazenamento; entre outros.

As atividades envolvidas nesta etapa observaram formalmente a execução dos passos da metodologia elencada para gestão do projeto, PMI/PMBoK.

A equipe da UnB considera que teve acesso a todas as informações necessárias à boa condução dos trabalhos e que a disponibilização dessas informações pela equipe da SE, assim como as atividades conjuntas de análise e discussão, levou a etapa do projeto a bom termo.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Elmasri, R; & Navathe, S.B. (2005). Sistemas de Banco de Dados. Editora: Pearson/Addison Wesley. Edição:4ª.ISBN: 8588639173.

[2] Takai, O.K.; Italiano, I.C; Ferreira, J.E. (2005). Introdução a Banco de Dados. DCC/IME/USP.

[3] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.

[4] Brewer, Eric A. "Towards robust distributed systems." PODC. Vol. 7. 2000.

[5] Cattell, Rick. "Scalable SQL and NoSQL data stores." ACM SIGMOD Record 39.4 (2011): 12-27.

[6] Chang, Fay, et al. "Bigtable: A distributed storage system for structured data." ACM Transactions on Computer Systems (TOCS) 26.2 (2008): 4.

[7] DeCandia, Giuseppe, et al. "Dynamo: amazon's highly available key-value store." ACM SIGOPS Operating Systems Review. Vol. 41. No. 6. ACM, 2007.

[8] Han, Jing, et al. "Survey on NoSQL database." Pervasive computing and applications (ICPCA), 2011 6th international conference on. IEEE, 2011.

[9] Angles, Renzo, and Claudio Gutierrez. "Survey of graph database models." ACM Computing Surveys (CSUR) 40.1 (2008): 1.

[10] Kaufman, C., Perlman, R; Speciner, M.(2002). Network Security: Private Communication in a Public World. Editora: Prentice Hall, Edição: 2ª. ISBN: 9780130460196.

[11] Goldreich, O. (2009). Foundations of Cryptography: Volume 2, Basic Applications (Vol. 2). Editora: Cambridge University Press.

[12] Song, D. X., Wagner, D., & Perrig, A. (2000). Practical techniques for searches on encrypted data. In Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on (pp. 44-55). IEEE.

[13] Gentry, C. (2009). A fully homomorphic encryption scheme. Tese de Doutorado. Stanford University.

Universidade de Brasília – UnB

Centro de Apoio ao Desenvolvimento Tecnológico – CDT

Laboratório de Tecnologias da Tomada de Decisão – LATITUDE

www.unb.br– www.cdt.unb.br – www.latitude.eng.br