5. MODELO DE BANCO DE DADOS NoSQL

5.1. INTRODUÇÃO

Um dos grandes desafios atualmente na área de Computação é a manipulação e processamento de grande quantidade de dados no contexto de **Big Data**. O conceito **Big Data** pode ser resumidamente definido como uma coleção de bases de dados tão complexa e volumosa que se torna muito difícil e complexa fazer algumas operações simples (remoção, ordenação, sumarização) de forma eficiente utilizando Sistemas Gerenciadores de Bancos de Dados (SGBD) tradicionais. Por causa desse problema, e outros demais, um novo conjunto de plataformas de ferramentas voltadas para **Big Data** tem sido proposto, como por exemplo: **Apache *Hadoop***.

Essa imensa quantidade de dados gerados traz novos grandes desafios na forma de manipulação, armazenamento e processamento de consultas em várias áreas de computação, e em especial na área de bases de dados, mineração de dados e recuperação de informação.

Nesse contexto, os SGBD tradicionais não são os mais adequados, ou "completos", às necessidades do domínio do problema de **Big Data**, como por exemplo: execução de consultas com baixa latência, tratamento de grandes volumes de dados, escalabilidade elástica horizontal, suporte a modelos flexíveis de armazenamento de dados, e suporte simples a replicação e distribuição dos dados.

Uma das tendências para solucionar os diversos problemas e desafios gerados pelo contexto **Big Data** é o movimento denominado **NoSQL** (*Not only SQL*). **NoSQL** promove diversas soluções inovadoras de armazenamento e processamento de grande volume de dados.

Estas soluções foram inicialmente criadas para solucionar problemas gerados por aplicações que em sua maioria necessitam operar com grande volume de dados, que tenham uma arquitetura que "escale" com grande facilidade de forma horizontal, que permitam fornecer mecanismos de inserção de novos dados de forma incremental e eficiente, além da necessidade de persistência dos dados em aplicações nas nuvens (*cloud computing*). Essas diversas soluções vêm sendo utilizadas com muita frequência em inúmeras empresas, como por exemplo, IBM, Twitter, Facebook, Google e Yahoo! para o processamento analítico de dados de *logs* Web, transações convencionais, etc.

Atualmente, existe uma grande adoção e difusão de tecnologias **NoSQL** nos mais diversos domínios de aplicação no contexto de **Big Data**. Esses domínios envolvem, em sua maioria, os quais os **SBGD tradicionais** ainda são *fortemente* dominantes como, por exemplo, instituições financeiras, agências governamentais, e comercio de produtos de varejo.

NoSQL não possui um significado padrão e pode ser traduzido em um conjunto de características, apresentadas a seguir:

- Não usa modelo de dados relacional e, portanto, não usa SQL;
- Costuma ser projetado para ser executado em um cluster;
- Costuma ser Open-Source;
- Não possui esquema fixo (SCHEMALESS), permitindo gravar qualquer dado em qualquer estrutura.

Schemaless é um recurso oferecido por SGBD que fazem uso da abordagem NoSQL para armazenar dados sem que seja necessário se importar com a normalização dos mesmos. Produtos como o **MongoDB e o Cassandra** fazem uso deste recurso para permitir que informações sejam armazenadas deixando toda a lógica de tratamento e organização da informação no lado da aplicação.

Utilizar **schemaless**, basicamente significa não se preocupar com os tipos de dados, nomes de colunas, nomes de tabelas e seus relacionamentos no formato relacional de um banco de dados, ou seja, não há necessidade de fazer normalização uma vez que a recuperação da informação pode ser realizada via endereçamento de referência simulando um vetor. Em teoria, o que este recurso permite é armazenar todos os dados sem o conhecimento prévio das chaves ou tipos de dados.

A informação é geralmente constituída por uma cadeia que representa a chave e os dados reais, e é nomeado de relacionamento "**chave-valor**". As informações em si são armazenadas em forma de algum dos tipos primitivos de uma linguagem de programação (uma *string*, um inteiro, um *array* ou um objeto) e são geridos por qualquer linguagem de programação capaz de realizar o gerenciamento do armazenamento das chaves e valores. A sugestão desta abordagem é de substituir a necessidade de modelo de dados fixo e tornar a exigência de dados formatados menos rigorosa.

5.1.1. Banco de Dados MongoDB

MongoDB é um banco de dados *schemaless*, orientado a documentos, open-source escrito em C++ que persiste documentos no formato BSON (Binary JSON), que são objetos JSON binários. **MongoDB** permite que usuários realizem modificações de apenas um atributo em um documento sem a necessidade de interação com o restante da estrutura.

Documentos podem ser armazenados em coleções, onde serão efetuadas operações de busca (*queries*) e indexação. *Queries* são expressas na sintaxe JSON e enviadas ao MongoDB como objetos BSON pelo driver de conexão ao banco de dados. Para persistência, o **MongoDB** usa arquivos mapeados em

memória, deixando o gerenciador de memória virtual do sistema operacional decidir quais partes vão para o disco e quais ficam na memória. Isto faz com que o **MongoDB** não tenha controle sobre o momento onde os dados são escritos no disco. **MongoDB** não possui controle de concorrência e gerenciamento de transações, diferentemente dos **Modelos Relacionais**.

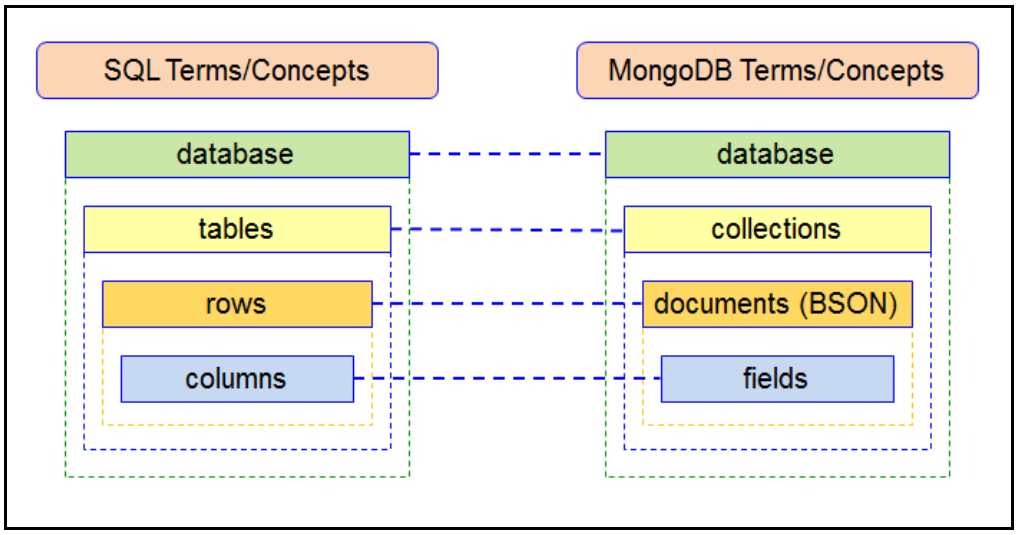

Figura 1. Modelo Relacional (SQL) x Modelo NoSQL

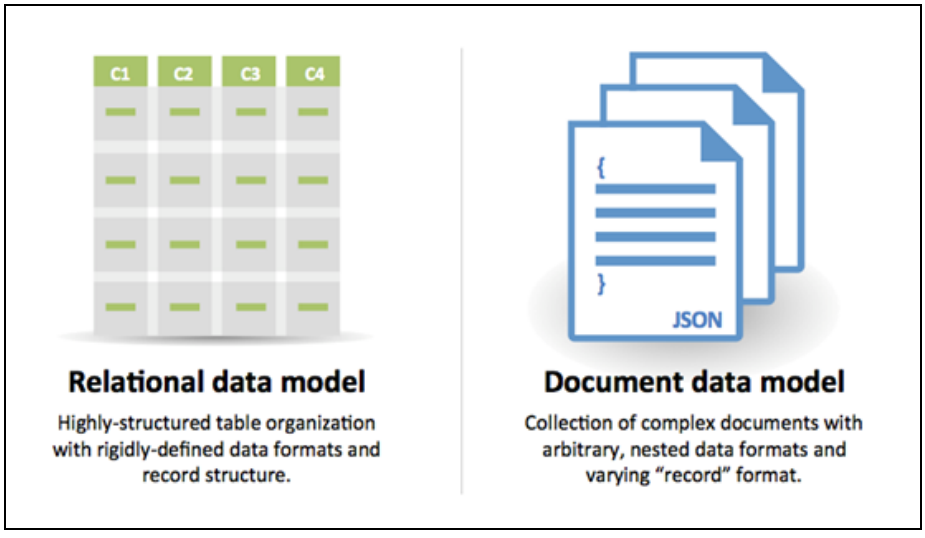

Figura 2. Modelo Relacional x Modelo em Documento

Figura 3. Ferramentas NoSQL e SQL

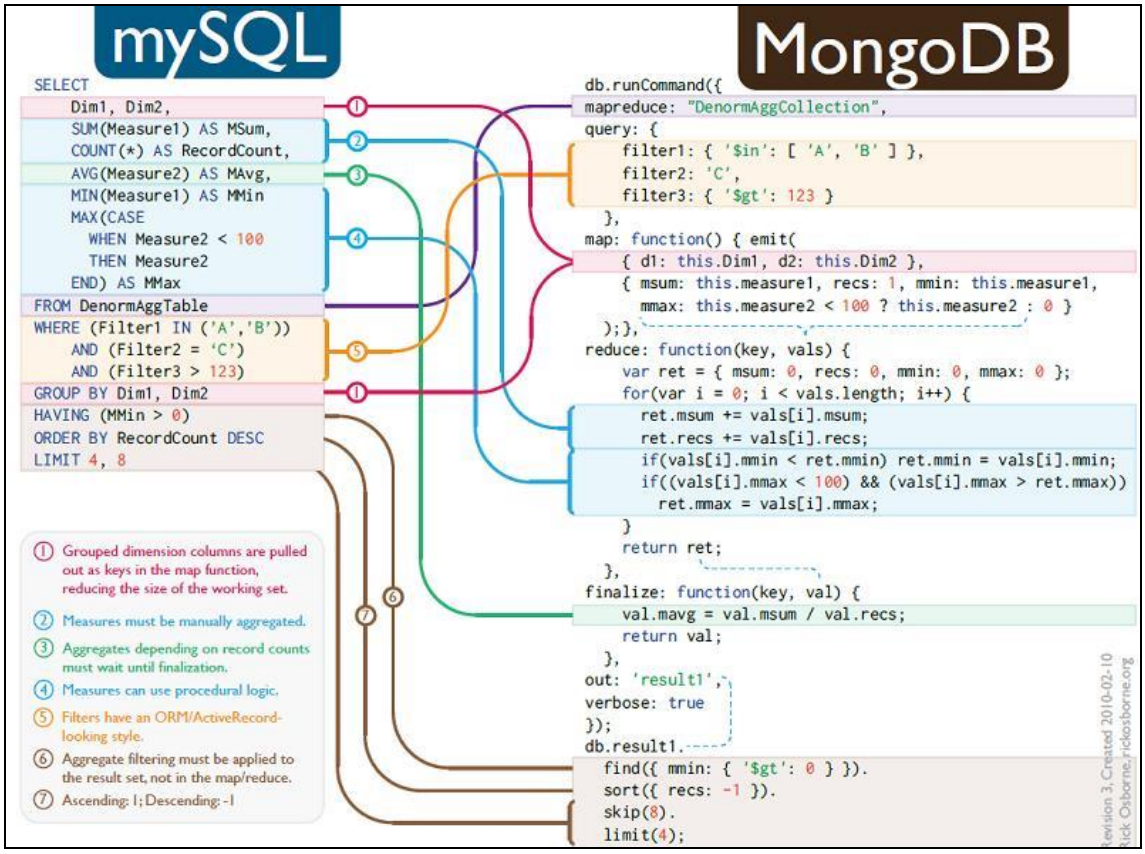

Figura 4. Código SQL x Código NoSQL

5.1.2. Comparação de Instruções SQL com instruções NoSQL

5.1.2.1. Criação, Alteração e Exclusão de Tabelas e Documentos

A instrução a seguir, criar uma tabela com base no MySQL (SQL), e um Documento com base no MongoDB (NoSQL).

CREATE TABLE people (id MEDIUMINT **NOT NULL** AUTO_INCREMENT, user_id Varchar(30), age Number, status Char(1), **PRIMARY KEY** (id))	db.people.insertOne({ user_id: "abc123", age: 55, status: "A" }) Ou criar explicitamente: **db.createCollection**("people")

A instrução a seguir, altera a tabela (**Table**) criada, com base no MySQL (SQL), e o Documento (**Collection**), com base no MongoDB (NoSQL).

As coleções não descrevem ou reforçam a estrutura de seus documentos; ou seja, não há alteração estrutural no nível de coleta. No entanto, no nível do documento, as operações **updateMany**()

ALTER TABLE people **ADD** join_date DATETIME	**Adicionar** campos (fields) aos documentos existentes usando o operador **$set**. db.people.updateMany({ }, { $set: { join_date: **new** Date() } })
ALTER TABLE people **DROP COLUMN** join_date	**Remover** campos (fields) de documentos usando o operador **$unset**. db.people.updateMany({ }, { $unset: { "join_date": "" } })

A instrução a seguir, exclui a tabela (**Table**) criada, com base no MySQL (SQL), bem como o Documento (**Collection**), com base no MongoDB (NoSQL).

DROP TABLE people	db.people.drop()

5.1.2.2. Inserção de dados em Tabelas e Documentos

As instruções a seguir, apresentam a inserção de linhas/registros em uma tabela do MySQL, bem como em uma coleção do MongoDB.

INSERT INTO people(user_id,age,status) **VALUES** ("bcd001",45,"A")	db.people.**insertOne**({ user_id: "bcd001", age: 45, status: "A" })

5.1.2.3. Consulta/Seleção de dados em Tabelas e Documentos

As instruções a seguir, apresentam consultas em tabela do MySQL, bem como em coleções do MongoDB.

SELECT * **FROM** people	db.people.**find**()
SELECT id, user_id, status **FROM** people	db.people.**find**({ }, { user_id: 1, status: 1 })
SELECT user_id, status **FROM** people **WHERE** status = "A"	db.people.**find**({ }, { user_id: 1, status: 1, _id: 0 })
SELECT * **FROM** people **WHERE** status = "A"	db.people.**find**({ status: "A" })

SELECT user_id, status **FROM** people **WHERE** status = "A"	db.people.**find**({ status: "A" }, { user_id: 1, status: 1, _id: 0 })
SELECT * **FROM** people **WHERE** status != "A"	db.people.**find**({ status: { $ne: "A" } })
SELECT * **FROM** people **WHERE** status = "A" **AND** age = 50	db.people.**find**({ status: "A", age: 50 })
SELECT * **FROM** people **WHERE** status = "A" **OR** age = 50	db.people.**find**({ $or: [{ status: "A" } , { age: 50 }] })
SELECT * **FROM** people **WHERE** age > 25	db.people.**find**({ age: { $gt: 25 } })
SELECT * **FROM** people **WHERE** age < 25	db.people.**find**({ age: { $lt: 25 } })
SELECT * **FROM** people **WHERE** age > 25 **AND** age <= 50	db.people.**find**({ age: { $gt: 25, $lte: 50 } })
SELECT * **FROM** people **WHERE** status = "A" **ORDER BY** user_id **ASC**	db.people.**find**({ status: "A" }).sort({ user_id: 1 })

SELECT * **FROM** people **WHERE** status = "A" **ORDER BY** user_id **DESC**	db.people.**find**({ status: "A" }).sort({ user_id: -1 })
SELECT COUNT(*) **FROM** people	db.people.**count**() *Ou* db.people.find().**count**()
SELECT COUNT(*) **FROM** people **WHERE** age > 30	db.people.**count**({ age: { $gt: 30 } }) *Ou* db.people.find({ age: { $gt: 30 } }).**count**()

5.1.2.4. Alteração de dados em Tabelas e Documentos

As instruções a seguir, apresentam a alteração de linhas/registros em uma tabela do MySQL, bem como em uma coleção do MongoDB.

UPDATE people **SET** status = "C" **WHERE** age > 25	db.people.**updateMany**({ age: { $gt: 25 } }, { $set: { status: "C" } })
UPDATE people **SET** age = age + 3 **WHERE** status = "A"	db.people.**updateMany**({ status: "A" } , { $inc: { age: 3 } })

5.1.2.5. Exclusão de dados em Tabelas e Documentos

As instruções a seguir, apresentam a exclusão de linhas/registros em uma tabela do MySQL, bem como em uma coleção do MongoDB.

DELETE FROM people **WHERE** status = "D"	db.people.**deleteMany**({ status: "D" })
DELETE FROM people	db.people.**deleteMany**({})

Para mais informações, acesse o manual em

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

Indubitavelmente, o crescimento da quantidade de dados e informação na Web é indiscutível e perceptível no nosso dia a dia. Entretanto, se as aplicações possuem um grande volume de dados é possível que se tenha grandes problemas em relação à infraestrutura. Atualmente, uma das abordagens mais adotadas é a distribuição horizontal (*scale out*), isto é, adicionar mais servidores para que atenda a aplicação. Entretanto, isso pode acarretar em um problema grande, já que, não é uma tarefa fácil efetuar a adição de um servidor em um sistema distribuído, devido a complexidade de configuração em uma aplicação que usa o **Banco de Dados Relacional** (TOTH, 2011).

Quando se trata de números relativamente grandes de registros (linhas), o desempenho pode ser prejudicado quando se usa uma abordagem de **Banco de Dados Relacional**, diminuindo a interação com o usuário.

O **NoSQL** surgiu para resolver essa problemática, mostrando uma abordagem diferente de persistência de dados, baseada em disponibilidade, desempenho e escalabilidade dos dados.

5.2. CATEGORIAS DE BANCOS DE DADOS NoSQL

Atualmente a diversidade de tipos de modelos e números de Banco de Dados **Não-Relacionais** (**NoSQL**) é grande, cada um possuindo conceitos e particularidades diferentes proporcionando ao desenvolvedor uma gama enorme, podendo atender necessidades distintas.

As principais categorias de **Bancos de Dados NoSQL** são:

- Orientados a documentos.
- Armazéns chave-valor.
- Orientados a colunas.
- Orientado a Grafos.

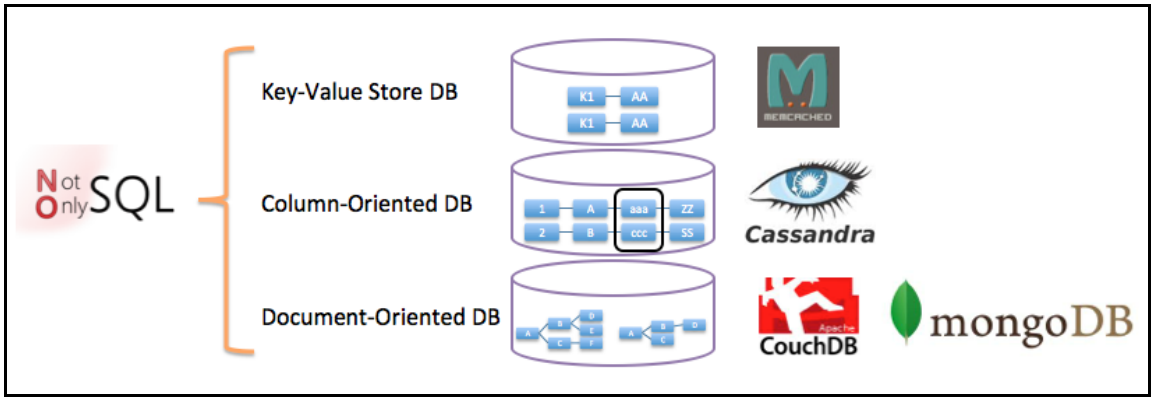

Figura 5. Ilustração das Categorias de Banco de Dados NoSQL

5.2.1. Bancos de dados orientados a documentos

O **CouchDB** é um dos bancos de dados orientado a documento amplamente usado no mercado, possuindo versões para web, desktop e mais recentemente para embarcados em plataformas móbile, como o Android e iOS.

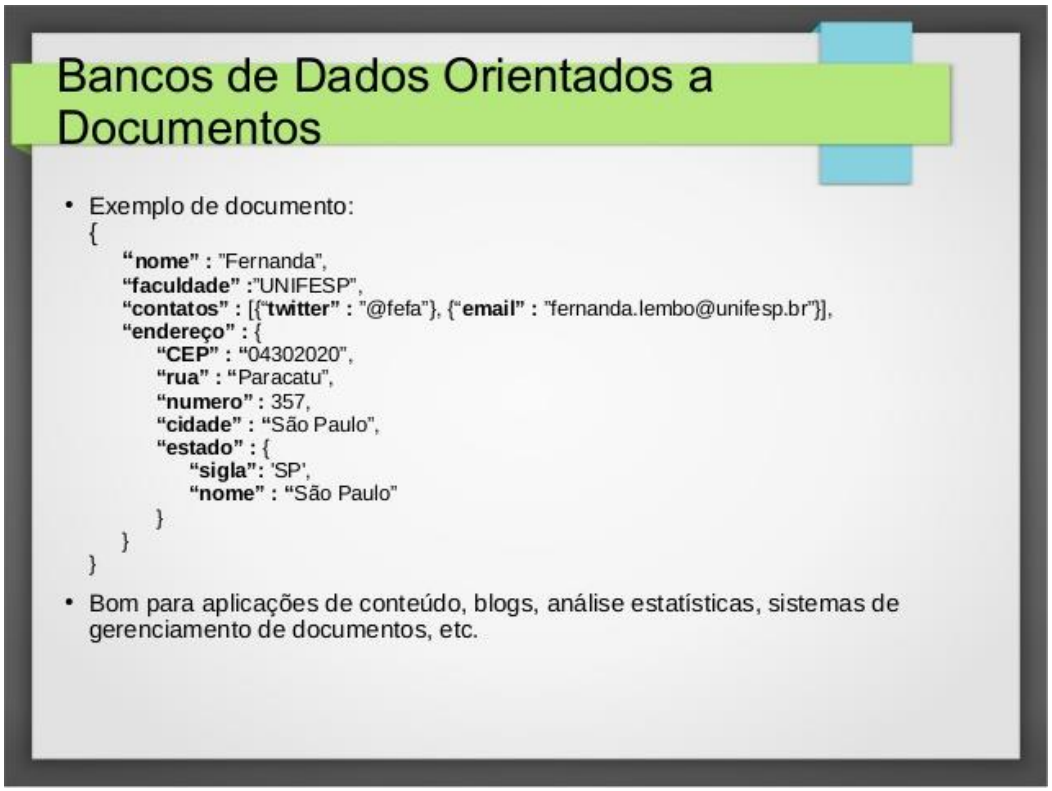

Figura 6. Exemplo de Código da Categoria Orientado a Documentos

Um banco de dados baseado em documento é, na sua essência, parecido com a abordagem chave/valor com uma grande exceção. Em vez de armazenar qualquer arquivo binário como valor de uma chave é requisitado que o valor dos dados que serão armazenados possua um formato que o banco possa interpretar. Na maioria das vezes podem ser arquivos de extensão XML, JSON, JSON-BLOB ou qualquer formato descritivo, variando muito de ferramenta para ferramenta.

5.2.2. Armazéns Chave-Valor

O modelo permite que o desenvolvedor efetue a persistência de dados totalmente livre de definições de estrutura para esquema. Como o nome já diz e sugere, trata-se de uma abordagem parecida com uma tabela *hash*. A informação do conteúdo é armazenada e um índice em forma de um tipo primitivo de dado é usado para mapeá-lo. O conteúdo pode ser armazenado em qualquer formato, seja uma *string*, inteiro, matriz ou objeto.

Pelo fato de ter uma chave de indexação para mapear um valor, a inserção e a recuperação se tornam mais fácil e com interface simplificada.

Na maioria dos casos a interface segue dois principais métodos (inserir e recuperar) que seguem o padrão:

- **Put**(chave, valor) – método que insere um registro
- **Get**(chave) – método que recupera um registro

Embora as vantagens de velocidade no armazenamento e recuperação de dados sejam indiscutíveis, é necessário considerar a quantidade de dados que esse modelo gera, uma vez que existem muita redundância e replicação de dados.

Um exemplo real pode se basear na necessidade de armazenar as lojas de um shopping Center, divididos em diversos tipos de departamentos, ocasionando muitos dados replicados para cada registro. Por exemplo, para cada registro de uma loja no shopping é necessário replicar o atributo departamento, sendo que em um modelo relacional pode-se trabalhar com identificadores e aplicar conceitos de normalização no esquema evitando que se repliquem os dados descritivos de uma entidade desnecessariamente.

Esse modelo é um pouco mais consolidado, uma vez que grandes empresas, como a própria Amazon, utilizam em seus sistemas, fornecem serviços, dentre outras ferramentas.

Uma ferramenta disponibilizada para o desenvolvedor é o Amazon's **SimpleDB**, o qual trata-se de um serviço na web que disponibiliza um banco de dados que usa o modelo de chave/valor com funções de indexamento para arquivos genéricos em uma nuvem.

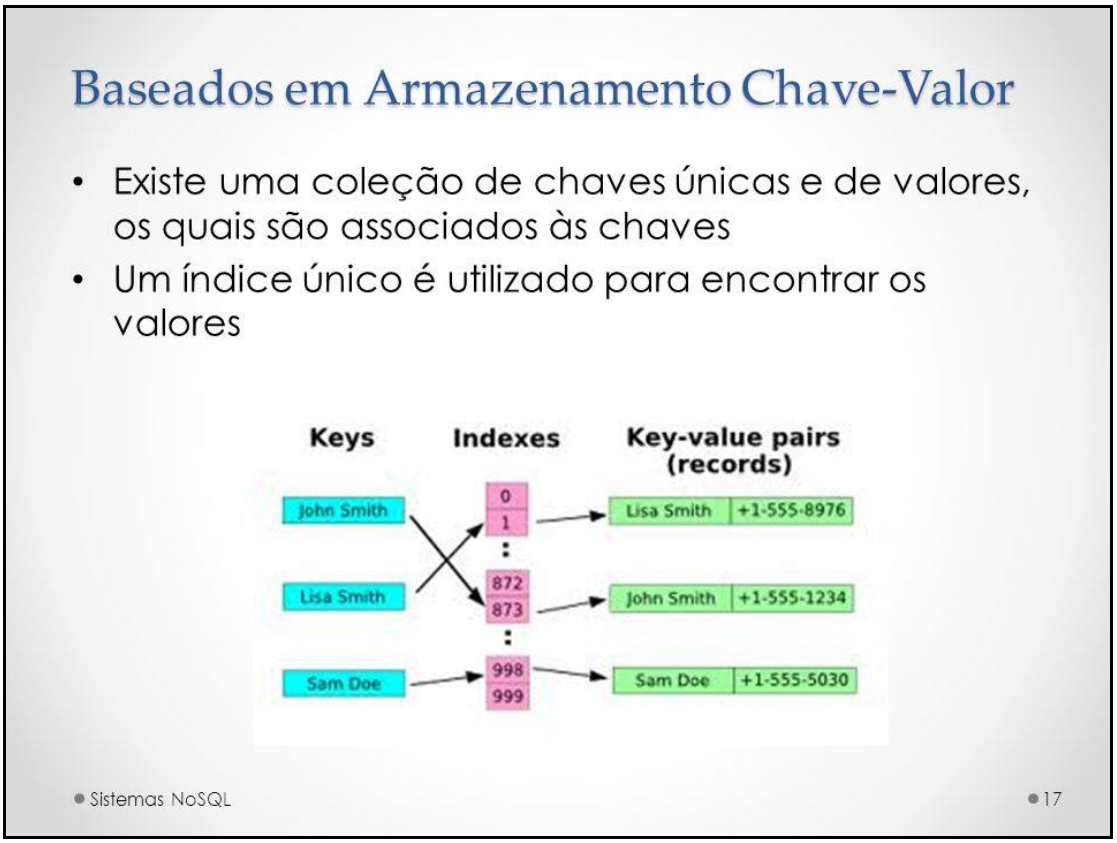

Figura 7. Ilustração da Categoria Armazém Chave-Valor

5.2.3. Bancos de Dados Orientados a Colunas

Com a crescente necessidade de análise de grandes volumes de dados no menor tempo possível, bancos de dados baseados em uso intensivo de memória RAM (***In-Memory***) estão cotados como sendo a solução mais viável. A principal diferença entre os bancos de dados In-Memory e os "tradicionais" está na forma de armazenamento dos dados, ***Column Store vs Row Store***. O primeiro lado da Figura 8, mostra o armazenamento em **Linhas** e o outro, o armazenamento em **Colunas**.

Figura 8. Ilustração da Categoria Orientado a Colunas

Analisando os exemplos, ao invés de cada registro da tabela ficar armazenado em uma linha, o registro passa a ser armazenado em colunas separadas. Essa forma de armazenamento tem algumas vantagens, como exemplo: a capacidade de compressão dos dados, visto que em banco de dados onde os registros são armazenados em linha, podem ser encontrados em uma mesma linha diferentes tipos (domínios) o que torna o processo mais complicado, já no banco orientado a colunas, cada coluna irá conter o mesmo tipo (domínio) de dado. De acordo com algumas pesquisas o nível de compressão alcançado em bancos orientados a colunas chega a ser de 60% a 70% mais eficiente que nos bancos orientados a linhas.

Para o caso de armazenamento dos dados em colunas é necessário que haja um relacionamento entre as colunas para identificar que a mesma pertence a um registro específico, para resolver esse problema a estratégia de incluir um ID virtual é normalmente adotada, nesse caso a representação do armazenamento em colunas ficaria conforme visto na Figura 9.

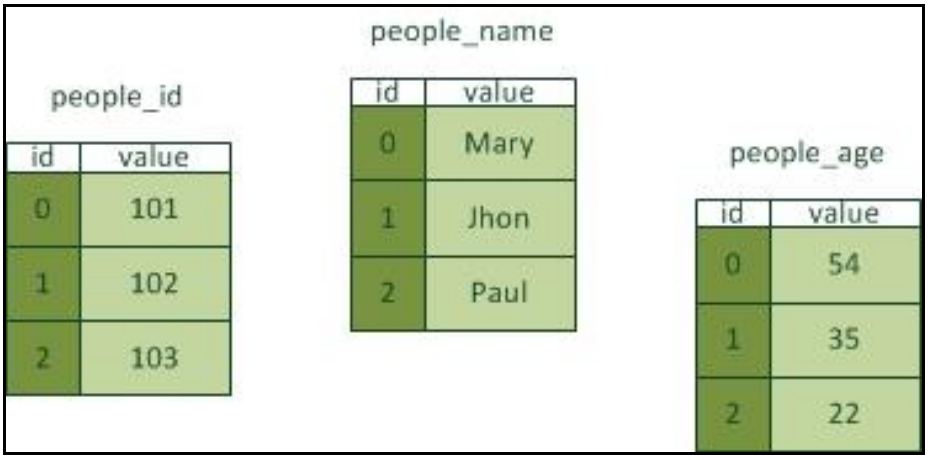

Figura 9. Ilustração da Categoria Orientado a Colunas com ID virtual

A query ***select avg(people_age) from people*** em um banco de dados orientado a linhas irá recuperar todas as linhas, carregando todos os campos (**colunas**) para executar a operação e retornar a média de idade, já no banco de dados orientado a colunas apenas a coluna **people_age** será avaliada consumindo assim menos recursos. Vale lembrar que um ***select * from people***, possivelmente não terá vantagem alguma já que todas as colunas precisarão ser lidas.

As principais técnicas aplicadas a banco de dados em coluna são: particionamento, indexação, compressão e diferentes estratégias para realização de *joins*.

5.2.4. Ferramentas

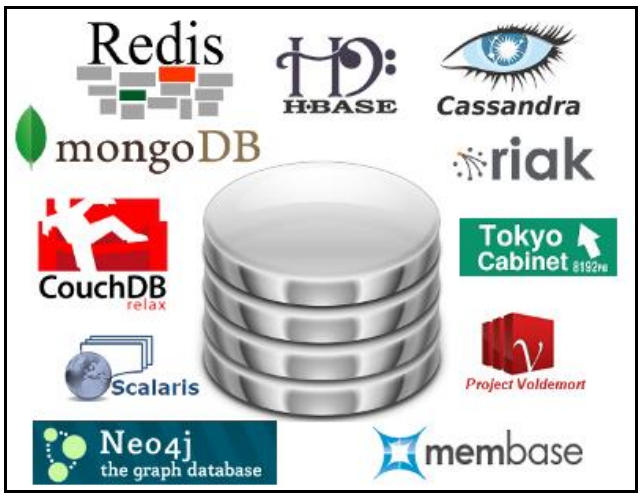

Figura 10. Ilustração de Ferramentas NoSQL

Graph	Column	Document	Persistent Key/Value	Volatile Key/Value
neo4j	BigTable (Google)	MongoDB (~BigTable)	Dynamo (Amazon)	memcached
FlockDB (Twitter)	HBase (BigTable)	CouchDB	Voldemort (Dynamo)	Hazelcast
InfiniteGraph	Cassandra (Dynamo + BigTable)	Riak (Dynamo)	Redis	
	Hypertable (BigTable)		Membase (memcached)	
	SimpleDB (AmazonAWS)		Tokyo Cabinet	

Figura 11. Ferramentas NoSQL por Categoria

5.3. ESCALABILIDADE EM BANCOS DE DADOS NoSQL

5.3.1. Bancos de Dados Relacionais

É verdade que a escalabilidade pode ser alcançada em banco de dados com o modelo relacional, entretanto, pode ser uma tarefa de custo elevado e uma tarefa bastante complexa.

Quando é necessário o aumento de infraestrutura para um banco de dados é normal usar como primeiro recurso uma **distribuição vertical (*scale up*)** de servidores, ou seja, quanto mais dados, aumenta-se a configuração física do equipamento (memória, processador e hard-disk). Essa pode ser uma solução excelente se tratando de curto prazo.

Outra possível solução é aplicar a técnica de **distribuição horizontal (*scale out*)**, ou seja, quanto mais dados, aumenta-se a quantidade de servidores, contudo, não necessariamente computadores com alto poder de processamento. A grande problemática dessa técnica é que a configuração para se integrar com os demais servidores já existentes no sistema distribuído com banco de dados relacional, podendo tornar o processo extremamente complexo, caro e demorado.

A técnica de distribuição vertical é limitada, então, deve-se usar a distribuição horizontal. Os bancos de dados relacionais a fim de garantir a integridade de dados utilizam o conceito **ACID** como propriedades em suas estruturas e transações.

As propriedades **ACID** (*Atomicity, Consistency, Isolation, Durability*) garantem que transações sejam feitas com segurança, garantindo atomicidade de cada uma delas, consistência dos dados, isolação e integridade durante todas as transações. Entretanto, garantir todos esses atributos pode acarretar em configurações complexas e um péssimo desempenho em uma transação ou consulta.

É sabido que as propriedades necessitam garantir os seguintes requisitos:

- Todos os nós da rede necessitam ter a mesma versão do arquivo (Consistência).
- Todos os clientes podem sempre encontrar pelo menos uma cópia dos dados solicitados, mesmo que um cluster estiver desligado (Disponibilidade).
- O sistema continua com seus dados, propriedades e características mesmo se for implantado em servidores diferentes, transparente para o cliente (Tolerância a partições).

5.3.2. Bancos de Dados Não-Relacionais

Para que seja possível trabalhar com um sistema distribuído foi proposto enfraquecer o requisito de consistência de dados, focando melhorar na disponibilidade e na tolerância para criar mais partições.

Foi dessa ideia que nasceu o conceito de **BASE** (*Basically Available, Soft state, Eventual Consistency*) com propriedades e filosofia oposta, provendo transações distribuídas, tolerância a falhas de consistência, e replicação otimista em um sistema distribuído.

A partir do conceito de **BASE** foram criados diferentes tipos de **Bancos de Dados Não Relacionais**, que devido à sua filosofia, não garantem consistência dos dados de uma aplicação.

Empresas grandes como Google e a Amazon aproveitam bastante a vantagem de BASE, utilizando um modelo **NoSQL**, aplicado ao conceito de distribuição horizontal (*scale out*).

Como não existe garantia de consistência, torna-se mais fácil efetuar configurações de particionamento no modelo e por consequência, este, torna-se mais escalável, e eficiente em questão de desempenho e de custo reduzido.

.

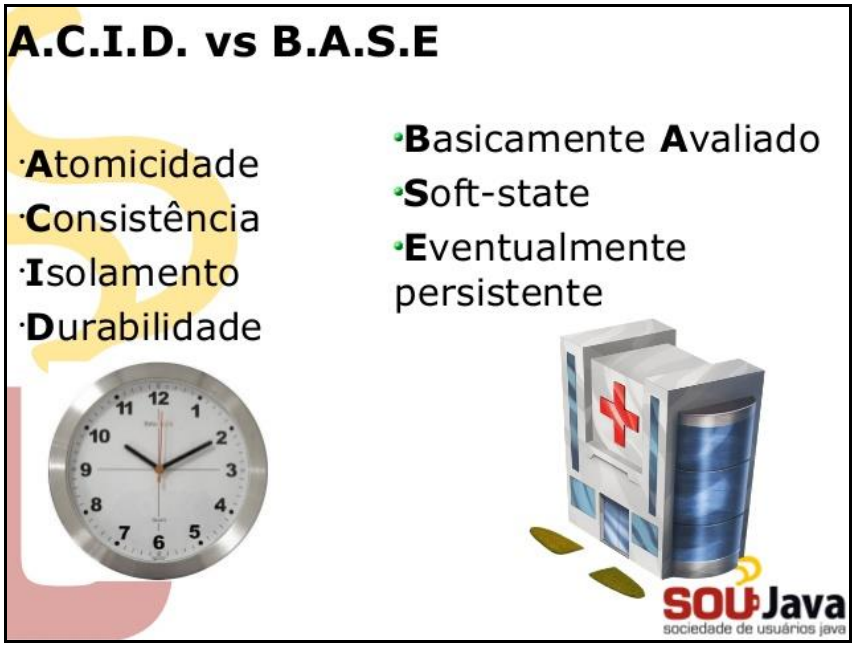

Figura 12. Ilustração ACID x BASE

5.3.3. Considerações Finais

Bancos de Dados NoSQL podem ser uma alternativa real para banco de dados que necessitam de alta escalabilidade, desempenho e que podem possuir tolerância a falhas. É importante perceber que o modelo proposto, não é uma Abordagem ideal que pode substituir todos os tipos de modelos de persistência, mas sim uma opção para as aplicações que possuem o cenário dentro das características discutidas.

Nos dias de hoje pode-se estabelecer um modelo ideal como um **sistema de banco de dados hibrido**, usando conceitos de diferentes abordagens sob demanda. Em uma aplicação, quando existe uma parte do sistema que necessita de consistência em seus dados, então é recomendado o uso de uma base de dados relacional, e quando necessitar de desempenho em um módulo, então usa-se o **NoSQL**.

Um exemplo real de modelo de dados híbrido foi implementado pela empresa **Boo-Box**. O modelo de negócios da companhia é baseado em propaganda integrada com sistemas de recomendação buscando se aproximar dos interesses do usuário, fazendo uma recomendação otimizada, baseada em conteúdo acessado.