Capítulo 2

Projeto de Bancos de Dados NoSQL

Angelo Augusto Frozza, Geomar André Schreiner and Ronaldo dos Santos Mello

Resumo

Bancos de Dados (BDs) NoSQL são uma tecnologia relativamente recente para gerenciamento de dados cuja principal motivação é um melhor atendimento dos desafios do Big Data, como volume, variedade e velocidade. Centenas de sistemas gerenciadores de BDs (SGBDs) NoSQL estão hoje disponíveis e alguns deles já ocupam posições elevadas no ranking de SGBDs populares mundialmente. NoSQL engloba uma família de modelos de dados não-relacionais: chave-valor, orientado a colunas, orientado a documentos e orientado a grafos. Considerando essa popularidade crescente e variedade de modelos de dados, torna-se pertinente a definição e adoção de uma metodologia de projeto para BDs NoSQL quando se deseja um esquema de base para a manipulação dos dados. Este capítulo apresenta uma visão geral dos modelos de dados NoSQL e uma proposta para projeto lógico de BDs NoSQL nos quatro modelos de dados supracitados, a partir de uma modelagem conceitual definida no modelo entidade-relacionamento estendido (modelo EER). Discute-se também a implementação deste projeto lógico em dois SGBDs NoSQL de amplo uso na indústria: MongoDB e Neo4j.

2.1. Apresentação

2.1.1. Objetivo

Este capítulo apresenta uma metodologia para o projeto lógico de BDs NoSQL. Em termos de projeto físico, considera-se dois SGBDs NoSQL: *MongoDB* e *Neo4j*.

2.1.2. Relevância para o Público-Alvo

A relevância deste capítulo se justifica pelo fato de BDs NoSQL [Sadalage and Fowler 2013, Atzeni et al. 2014] serem uma tecnologia relativamente recente para o gerenciamento de dados e, desta forma, os cursos de graduação na área de Ciência da Computação geralmente não apresentam essa tecnologia em suas disciplinas da área de BD. Disciplinas na área de BD enfatizam o modelo de dados relacional, porém, NoSQL compreende uma

família de novos modelos de dados não-relacionais que vêm crescendo em termos de utilização pela indústria. Assim sendo, ao final deste capítulo o público-alvo terá adquirido um conhecimento inovador na área de BD.

Além disso, o projeto de BDs NoSQL é um tópico pouco explorado na literatura de BD. Por este motivo, o conteúdo desse capítulo se torna também interessante para alunos de pós-graduação, pesquisadores e profissionais entusiastas que buscam inovação na gestão dos seus dados. Os autores deste capítulo têm trabalhado em pesquisas relacionadas a este tópico nos últimos anos [Lima and Mello 2015, Lima and Mello 2016, Schreiner et al. 2019, Frozza and Mello 2020, Santana and Mello 2020, Schreiner et al. 2020, Silva and Mello 2021].

2.1.3. Autores

Os autores deste capítulo são todos professores com vários anos dedicados ao ensino e pesquisa na área de BD. Seus curriculos resumidos são os seguintes:

- **Angelo Augusto Frozza** é doutor em Ciência da Computação pela Universidade Federal de Santa Catarina (UFSC) e professor efetivo no Instituto Federal Catarinense (IFC), Campus Camboriú. Sua área de atuação é BD e seus principais tópicos de pesquisa são data warehouse, BDs geográficos e modelagem de dados. CV Lattes: *http://lattes.cnpq.br/5878372087019892*.
- **Geomar André Schreiner** é doutor em Ciência da Computação pela UFSC e professor na Universidade do Oeste de Santa Catarina (UNOESC) em Chapecó. Sua área de atuação é BD e seus principais tópicos de pesquisa são BDs NoSQL, BDs NewSQL, interoperabilidade de dados e particionamento de dados. CV Lattes: *http://lattes.cnpq.br/0776438722468291*.
- **Ronaldo dos Santos Mello** é doutor em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS) e professor titular na UFSC. Coordena o Grupo de pesquisa em BD da UFSC e o PET Computação da UFSC. Sua área de atuação é BD e seus principais tópicos de pesquisa são modelagem de dados, restrições de integridade, integração e interoperabilidade de dados, BD NoSQL e BD NewSQL. CV Lattes: *http://lattes.cnpq.br/5011370918857999*.

2.1.4. Organização do Capítulo

Este capítulo está organizado em mais 4 seções. A Seção 2 apresenta uma visão geral de BDs NoSQL. A Seção 3 detalha uma metodologia para projeto lógico de BDs NoSQL e a Seção 4 exemplifica a implementação de um projeto lógico nos SGBDs MongoDB e Neo4j. Por fim, a Secão 5 é dedicada às considerações finais.

2.2. Introdução a BDs NoSQL

Esta seção apresenta a origem dos BDs NoSQL, suas principais propriedades e modelos de dados. O texto tem por base o livro *NoSQL Distilled* [Sadalage and Fowler 2013].

2.2.1. Um Pouco de História

O termo "NoSQL"foi utilizado pela primeira vez em 1998, por Carlos Strozzi, para apresentar sua proposta de BD relacional de código aberto que não implementava a linguagem SQL[1]. Porém, esse SGBD não teve nenhuma influência sobre o desenvolvimento dos BDs que seguem o paradigma NoSQL.

No final dos anos 2000, a Big Table, do Google, e o DynamoDB, da Amazon, inspiravam o desenvolvimento de inúmeros projetos alternativos para armazenamento de dados, sendo que isso tornou-se assunto nas melhores conferências de software da época.

Em 2009, Johan Oskarsson, um desenvolvedor de software de Londres, foi a São Francisco (EUA) para um evento sobre *Hadoop*. Ele também estava interessado em conhecer mais sobre esses novos SGBDs não relacionais, distribuídos e de código aberto que estavam surgindo e, como estava com pouco tempo, resolveu organizar uma reunião para que todos pudessem apresentar seus trabalhos para quem estivesse interessado em conhecê-lo. Assim, participaram representantes dos projetos Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB e MongoDB. Johan procurava um nome para identificar essa reunião, o qual teria que ser uma boa *hashtag* para o *Twitter* (curto, fácil de lembrar e sem muitos similares no Google). Ele usou seus contatos para pedir sugestões e, entre as recebidas, escolheu usar "*NoSQL Meetup*".

Atualmente, o termo NoSQL é aplicado apenas em SGBDs desenvolvidos a partir do século XXI e que possuem as principais características dos BDs NoSQL. Para [Sadalage and Fowler 2013], o termo NoSQL é entendido como "*Not only SQL*" e remete a SGBDs que não seguem o modelo de dados relacional ao mesmo tempo que não pretendem substituir o modelo relacional, mas sim, ser uma opção quando é necessária uma maior flexibilidade na estruturação do BD. Michael Stonebreaker cita duas razões principais que motivam a troca de um SGBD relacional por um SGBD NoSQL: *(i)* necessidade de um *desempenho* melhor; e *(ii)* necessidade de maior *flexibilidade* [Stonebraker 2010].

Os BDs NoSQL foram projetados para gerenciar grandes volumes de dados e apresentam 5 propriedades principais [Schreiner et al. 2015]: *(i)* Escalabilidade horizontal; *(ii)* Armazenamento distribuído; *(iii)* Interface de acesso simples; *(iv)* Alta disponibilidade; e *(v)* Esquemas flexíveis ou mesmo ausência de esquema.

2.2.2. Modelos de dados NoSQL

BDs NoSQL podem ser categorizados pelo modelo de dados adotado: *chave-valor, orientado a documentos, orientado a colunas* e *orientado a grafos*. Os três primeiros são conhecidos como *modelos NoSQL baseados em chave*, pois compartilham o princípio de recuperar dados a partir de uma chave de entrada, enquanto diferem em termos de acesso aos componentes internos dos dados [Atzeni et al. 2014]. Esses modelos de dados são

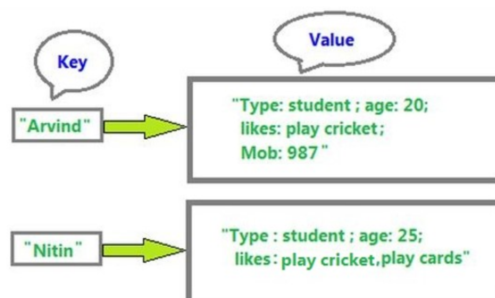
[1]NoSQL Relational Database Management System,`http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page`

detalhados a seguir.

2.2.2.1. Chave-valor

O modelo de dados *chave-valor* é o mais simples, similar a uma estrutura de indexação, como uma *hash table*. Ele mantém um conjunto de pares chave-valor, sendo o valor acessado por meio da chave única (ver Figura 2.1). Um valor pode conter um conteúdo simples ou complexo, mas esse conteúdo é tratado como um componente único, ou seja, uma “caixa-preta". Devido a isso, assume-se que qualquer valor no modelo de dados chave-valor tem um domínio atômico [Sadalage and Fowler 2013, Atzeni et al. 2014].

Figura 2.1. Representação do modelo de dados chave-valor



A API de acesso é simples, contendo as operações *get(key)*, *put(key, value)* e *delete(key)*. Não suporta definição de esquemas, relacionamentos entre os dados e não possui uma linguagem de consulta. Exemplos de SGBDs NoSQL chave-valor populares são o *Redis* e o *Voldemort*.

2.2.2.2. Orientado a documentos

O modelo *orientado a documentos* define uma coleção de documentos, sendo cada documento acessado por uma chave única. Um documento é composto por atributos atômicos ou atributos complexos com conteúdo estruturado principalmente a partir do uso dos construtores JSON para *objeto* e *array* que podem, por sua vez, abrigar valores atômicos ou complexos (ver Figura 2.2) [Sadalage and Fowler 2013, Atzeni et al. 2014].

O modelo orientado a documentos é mais adequado para a representação de objetos complexos. SGBDs que seguem esse modelo possuem APIs de acesso proprietárias e/ou linguagens de consulta simples. Não suporta relacionamento entre dados. Exemplos de SGBDs NoSQL orientado a documentos são o *MongoDB* e o *CouchDB*.

2.2.2.3. Orientado a colunas

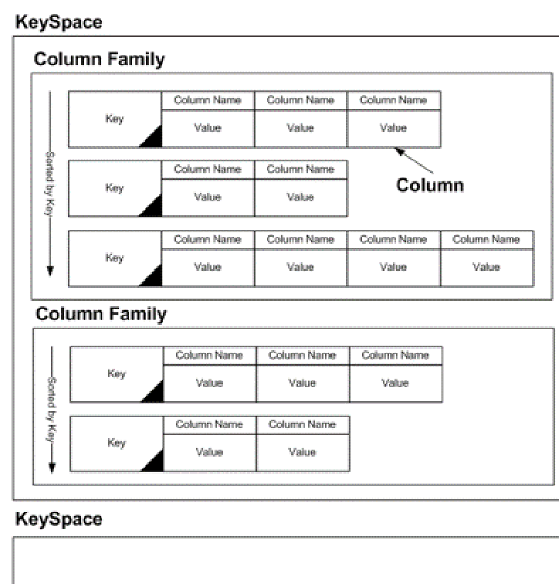
O modelo de dados *orientado a colunas* organiza os dados com base em um modelo distribuído em colunas, semelhante ao modelo relacional, mas com esquema flexível. Novas colunas podem ser definidas no momento do armazenamento. Por causa disso,

Figura 2.2. Exemplo de um documento (formato JSON)

```
{ "_id":"discussion_tables",
  "_rev":"D1C946B7",
  "Sunrise":true,
  "Sunset":false,
  "FullHours":[1,2,3,4,5,6,7,8,9,10],
  "Activities": [
     {"Name":"Football", "Duration":2, "DurationUnit":"Hours"},
     {"Name":"Breakfast", "Duration":40, "DurationUnit":"Minutes",
           "Attendees":["Jan", "Damien", "Laura", "Gwendolyn", "Roseanna"]}
   ]      }
```

instâncias de dados de uma mesma entidade podem ter um número diferente de colunas e alguns BDs dessa classe suportam o conceito de supercoluna. Ao mesmo tempo, colunas são agrupadas em famílias, o que auxilia o particionamento vertical e a otimização física (ver Figura 2.3). A criação de novas tabelas e novas famílias de colunas só é possível em tempo de projeto de esquema, quando os dados não podem ser acessados [Atzeni et al. 2014].

Figura 2.3. Representação do modelo orientado a colunas.

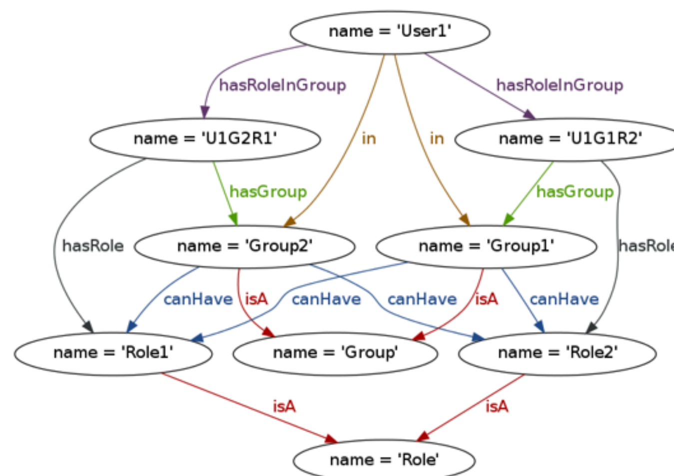


Comparando com BDs relacionais, o conceito de *keyspace* é equivalente ao conceito de *BD*, uma *família de colunas* é equivalente a uma *tabela* e um *conjunto de colunas* é equivalente a um *registro*. Uma coluna possui um nome e um valor, um conjunto de colunas é acessado por uma chave e instâncias de dados (registros) podem ter quantidade de colunas diferentes. Alguns SGBDs possuem suporte a colunas multivaloradas e supercolunas. O acesso aos dados é por meio de APIs proprietárias e/ou linguagens de consultas simples e não suportam relacionamentos entre dados. Alguns exemplos de SGBDs NoSQL orientados a colunas são o *Cassandra* e o *HBase*.

2.2.2.4. Orientado a grafos

O modelo de dados *orientado a grafos* consiste basicamente de vértices conectados por arestas [Sadalage and Fowler 2013]. Essa estrutura é indicada para armazenar pequenos registros que possuem interconexões complexas, como é o caso de dados em redes sociais. Um vértice representa um objeto do mundo real, enquanto uma aresta representa uma relação entre dois vértices. Propriedades podem ser atribuídas para vértices e arestas e são formadas por pares chave-valor, sendo que o valor é restrito a um determinado tipo de dados (Figura 2.4).

Figura 2.4. Representação do modelo orientado a grafos



O acesso aos dados é possível através de APIs proprietárias e/ou linguagens de consultas simples. Exemplos de SGBDs NoSQL orientados a grafos são o *Neo4j* e o *InfiniteGraph*.

2.3. Projeto lógico de BDs NoSQL

2.3.1. Projeto clássico de BD

O projeto de um BD é parte integrante do desenvolvimento de um sistema informatizado que necessita lidar com um grande volume de dados. Seus principais objetivos são a representação adequada dos dados operacionais da Organização e o armazenamento/acesso eficientes a estes dados [Henry Korth and Silberschatz 2019]. Uma metodologia clássica de projeto de BD apresenta 3 etapas que definem esquemas de dados em diferentes níveis de abstração a partir do levantamento dos requisitos de dados da Organização [Heuser 2009, Elmasri and Navathe 2011]:

- *projeto conceitual*: modelagem de dados de alto nível de abstração, ou seja, independente do(s) modelo(s) de dados do(s) SGBD(s) alvo. Seu objetivo é o entendimento dos dados e seus relacionamentos dentro da Organização, bem como a validação deste entendimento junto aos usuários da Organização. O modelo clássico utilizado para a especificação de um esquema conceitual é o modelo Entidade-Relacionamento Estendido (modelo EER);
- *projeto lógico*: conversão do esquema conceitual em esquema(s) válido(s) no(s) modelo(s) de dados do(s) SGBD(s) alvo, como por exemplo, o modelo relacional e o modelo orientado a documentos. Nesta etapa aplicam-se regras de conversão visando gerar esquema(s) de dados eficientes para fins de armazenamento, assim como acesso pelas principais operações de manipulação de dados;
- *projeto físico*: esta etapa realiza a especificação do(s) esquema(s) lógico(s) no(s) SGBD(s) alvo, ou seja, realiza-se a implementação das estruturas de dados e dos recursos para garantia de integridade, segurança e otimização de acesso utilizando a linguagem de definição de dados (DDL) do(s) SGBD(s). Exemplos no contexto de BDs relacionais são a definição de tabelas, índices, visões e gatilhos.

A Figura 2.5 mostra uma modelagem conceitual utilizando o modelo EER (diagrama à esquerda) e uma possível modelagem lógica relacional (diagrama à direita) para o domínio de um *Museu*. Fatos do mundo real (ou entidades) são representados por retângulos nomeados (p.ex., *Obras*) enquanto relacionamentos entre esses fatos são representados por losangos nomeados (p.ex., *exposição*) e possuem cardinalidades associadas (p.ex., uma obra pode estar exposta em um único salão e um salão pode abrigar várias obras). Entidades e relacionamentos podem conter atributos (p.ex, *título* da obra e *posição* da exposição, respectivamente), sendo possível definir atributos compostos (p.ex., *endereco*), multivalorados (p.ex., *diasAtendimento*) e identificadores (p.ex., *CNPJ* do fornecedor). Ainda, uma entidade pode se especializar em uma ou mais entidades que podem apresentar propriedades adicionais (atributos e/ou relacionamentos), como é o caso de uma obra, que pode ser uma *Pintura* ou uma *Escultura*. Por fim, uma entidade associativa (p.ex., *restauração*) denota um relacionamento que é promovido a uma entidade para que possa se relacionar com outras entidades.

A modelagem lógica relacional, por sua vez, é gerada através da aplicação de regras de conversão. Basicamente, uma entidade se torna uma tabela (p.ex., *Pais*) e

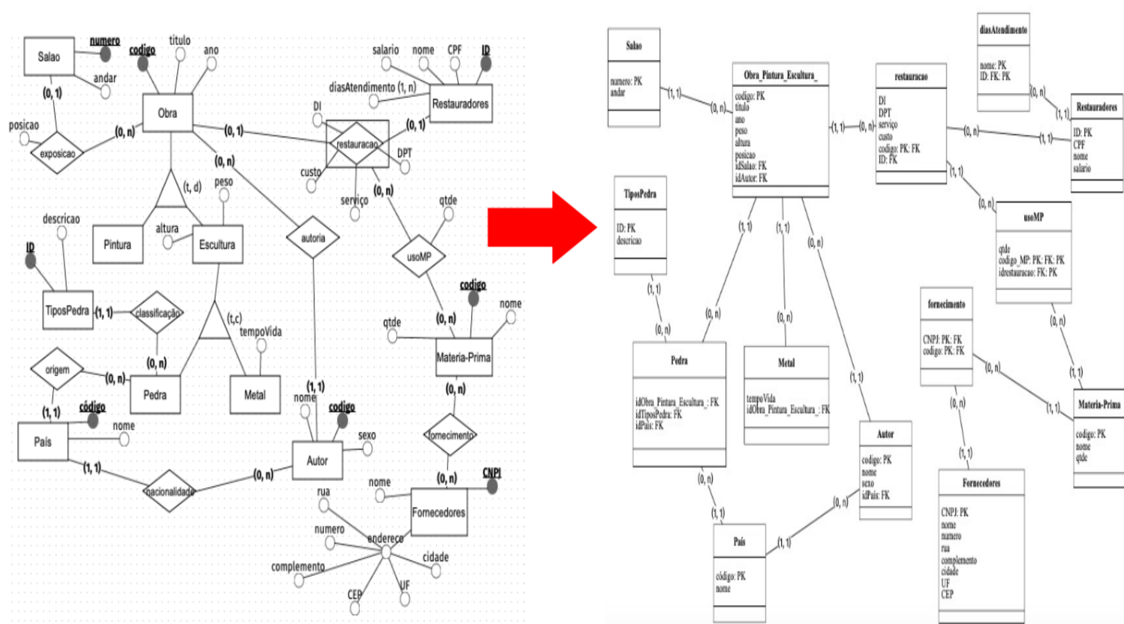


Figura 2.5. Exemplo de modelagem conceitual e uma correspondente modelagem lógica

um relacionamento pode ser representado por uma chave estrangeira (p. ex., *idTipoPedra* em Pedra), gerar uma tabela própria (p.ex., *usoMP*) ou gerar uma tabela resultante da sua fusão com os dados das entidades envolvidas. Entidades envolvidas em um relacionamento de especialização podem ser fundidas em uma única tabela (p.ex., *Obras_Pinturas_Esculturas*), gerar tabelas para cada entidade ou ainda tabelas apenas para as entidades especializadas (p.ex., *Pedra* e *Metal* no caso de Esculturas). Atributos multivalorados e entidades associativas geralmente geram tabelas.

A Figura 2.6 exemplifica parte da especificação de um projeto físico para a modelagem lógica apresentada na Figura 2.5 utilizando a linguagem DDL da SQL. Ilustra-se a definição da tabela *Salao* com seus atributos e restrições, a definição de um índice para acelerar buscas por *cidade* na tabela *Fornecedores* e uma visão que permite visualizar dados de obras e autores conjuntamente.

2.3.2. Adequação do Projeto Clássico de BD a BDs NoSQL e o Modelo de Agregados

Conforme descrito na Seção 2.2.2, os BDs NoSQL são uma família de modelos lógicos de dados. Assim sendo, a etapa de projeto lógico deve ser capaz de converter um esquema EER para um ou mais desses modelos NoSQL. Para evitar uma quantidade excessiva de regras de mapeamento para cada modelo NoSQL, uma estratégia para minimizar a complexidade do projeto lógico pode ser a adoção de um modelo intermediário que seja capaz de abstair características comuns de mais de um desses modelos NoSQL.

Neste trabalho consideramos o modelo de agregados, conforme sugerido por Sadalage & Fowler [Sadalage and Fowler 2013], como modelo intermediário. Desta forma, o projeto lógico clássico de um BD pode ser desmembrado em 2 etapas: *(i) projeto lógico de alto nível*; e *(ii) projeto lógico de baixo nível*. A primeira converte um esquema EER para um esquema baseado no modelo de agregados e a segunda converte o esquema

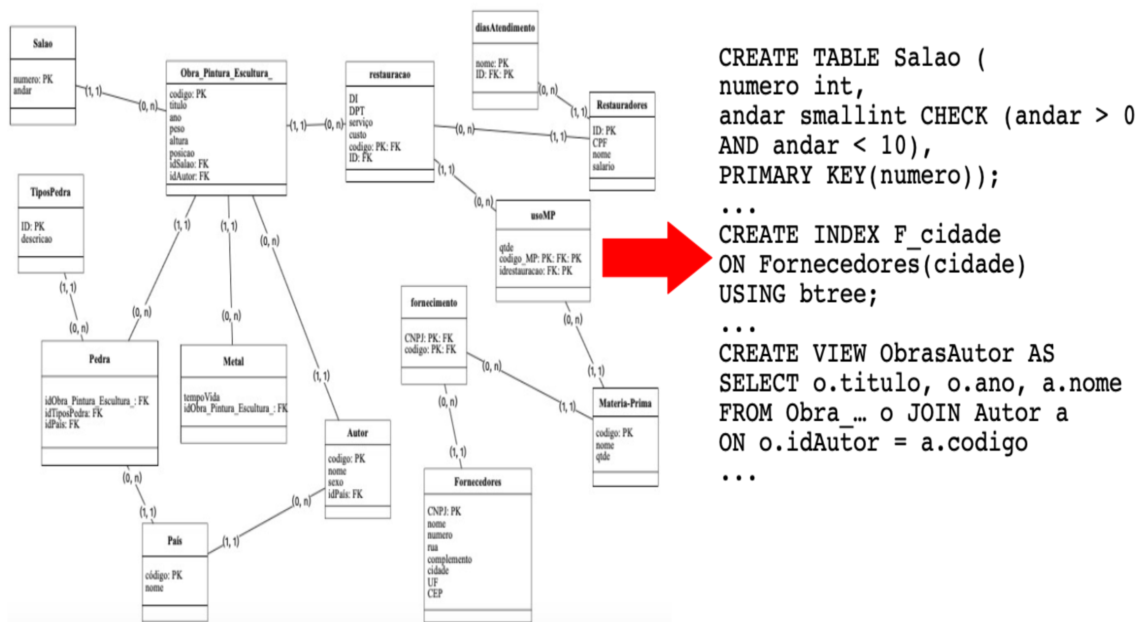


Figura 2.6. Exemplo de mapeamento de uma modelagem lógica para uma modelagem física

de agregados (ou parte dele, caso seja desejada a persistência do esquema conceitual em mais de um BD NoSQL) para o esquema de um BD NoSQL.

O modelo de agregados é adequado à abstração de dados de alto nível para 3 modelos de dados NoSQL: orientado a documentos, orientado a colunas e chave-valor. Isso por que o modelo de agregados é capaz de representar objetos complexos que possuem uma chave de acesso (identificador). Essas características de representação estão presentes nesses 3 modelos.

O modelo de agregados foi formalizado por Lima & Mello [Lima and Mello 2015]. Neste trabalho também foi definida uma notação gráfica para ele, conforme exemplificado na Figura 2.7 para dados no domínio de um Museu.

Os conceitos do modelo de agregados são: *(i) coleção; (ii) bloco; (iii) atributo; (iv) relacionamento de agregação; (v) relacionamento de referência;* e *(vi) restrição de disjunção*. O primeiro representa um objeto de um mundo real (geralmente uma entidade) e possui um identificador (atributo com prefixo *ID_*). Uma coleção pode ser referenciada por outra (ou pela mesma) coleção. Exemplos são *Obra* e *Salão* na Figura 2.7.

Um bloco é um componente interno a uma coleção ou a outro bloco, e pode ter uma cardinalidade associada, indicando que várias ocorrências dele podem existir. Um exemplo na Figura 2.7 é o bloco opcional *autoria*. Tanto blocos quanto coleções podem ter atributos que descrevem seus dados, podendo eles ser mono ou multivalorados. Um exemplo é o atributo monovalorado *nacionalidade* no bloco autoria e o atributo multivalorado *horários* na coleção Salão.

Os dois tipos de relacionamento presentes no modelo de agregados são agregação e referência. O primeiro é representado pelos blocos aninhados em coleções ou outros blocos, indicando a existência de um objeto interno a outro, como é o caso de *Pintura*

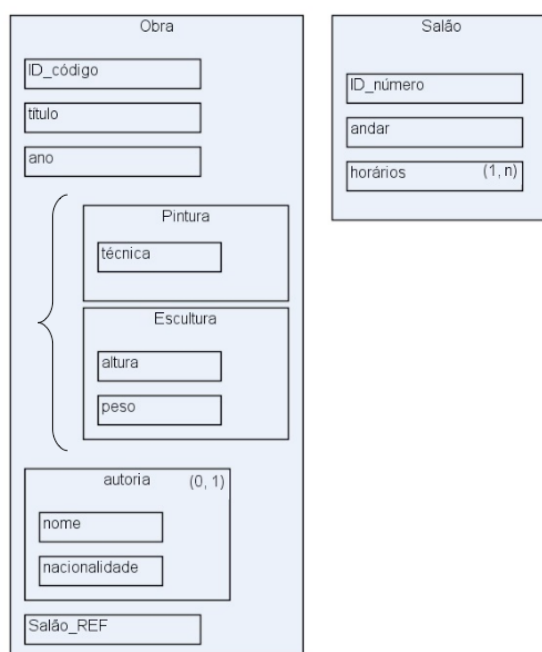


Figura 2.7. Exemplo de esquema de dados representado no modelo de agregados

e *Escultura* na coleção *Obra*. O segundo representa relacionamentos entre coleções e é definido por um atributo com sufixo *_REF*. Um exemplo na Figura 2.7 é o atributo *Salao_REF*, indicando o salão onde a obra pode estar exposta.

Por fim, uma restrição de disjunção indica blocos mutuamente exclusivos em uma coleção, sendo uteis para representar especializações disjuntas em um diagrama EER. Blocos mutuamente exclusivos são denotados no modelo de agregados por um ”abre chave” envolvendo todos os blocos que devem ser disjuntos. Um exemplo na Figura 2.7 são os blocos *Pintura* e *Escultura* na coleção *Obra*.

A seguir detalha-se as duas etapas do projeto lógico de um BD NoSQL. Elas foram propostas no trabalho de Lima [Lima 2016].

2.3.3. Projeto Lógico de Alto Nível para BDs NoSQL

A etapa de projeto lógico de alto nível converte um esquema conceitual representado no modelo EER para um esquema de agregados. Ela executa 2 subetapas, nesta ordem: *(i) conversão de hierarquias; (ii) conversão de relacionamentos*. Estas conversões tomam como base as regras de mapeamento para projeto lógico de BDs relacionais [Heuser 2009].

A primeira subetapa analisa cada hierarquia de especialização presente no esquema EER, através de um percorrimento *bottom-up* dos seus níveis hierárquicos, e aplica uma dentre as seguintes regras de conversão para cada nível:

1. *ênfase na entidade genérica*: gera-se uma única coleção para representar o nível hierárquico. O nome da coleção é o nome da entidade genérica e seus atributos são os atributos de todas as entidades presentes no nível. Um atributo especial *tipo*

é definido para indicar qual(is) entidade(s) do nível (genérica e/ou especializadas) está(ão) sendo instanciada(s)[2]. Um exemplo é mostrado na Figura 2.8 (a);

2. *ênfase nas entidades especializadas*: gera-se uma coleção para cada entidade especializada. Seus atributos incluem os atributos da entidade genérica. Um exemplo é mostrado na Figura 2.8 (b);

3. *ênfase na hierarquia*: gera-se uma única coleção para todo o nível representando a entidade genérica, bem como blocos aninhados a esta coleção representando as entidades especializadas. Caso a especialização seja disjunta, uma restrição de disjunção é definida. Um exemplo é mostrado na Figura 2.8 (c).

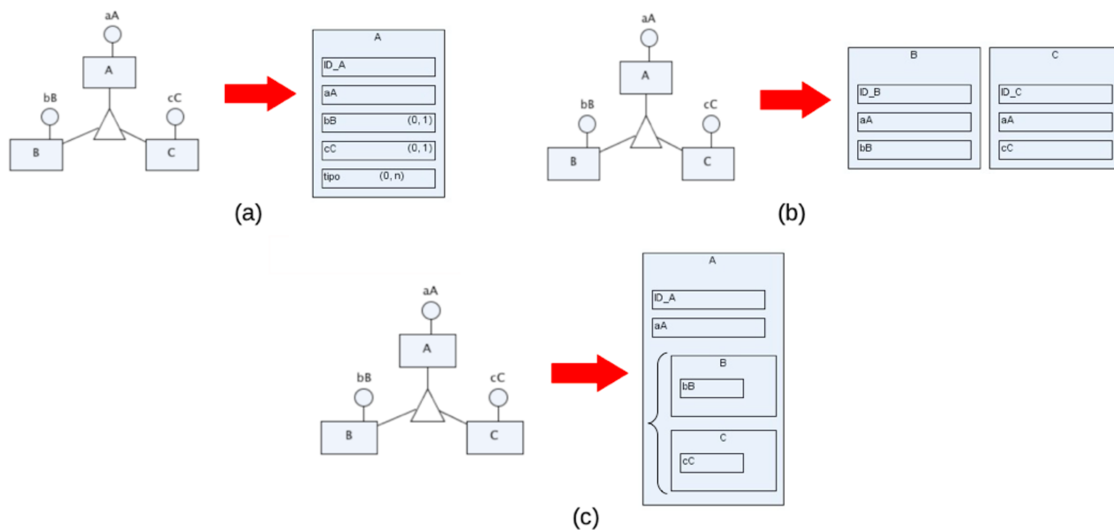


Figura 2.8. Conversão de hierarquias em um esquema EER

A escolha por uma ou outra alternativa depende de fatores como o *tamanho do esquema*, a *carga de trabalho* típica (*workload*) da aplicação e a quantidade de *restrições de integridade* que devem ser definidas para respeitar o esquema gerado. Por exemplo, a alternativa 1 gera um esquema mais enxuto e menos complexo, sendo mais adequada a situações onde a maioria das instâncias no nível hierárquico são da entidade genérica ou quando a maioria dos atributos e relacionamentos pertencem à entidade genérica. Entretanto, ela requer alguns controles de integridade associados ao tipo da instância. Conforme ilustra a Figura 2.8 (a), se o tipo da instância for B, então o atributo *bB* deve ter valor enquanto que o atributo *cC* deve ser NULL. A alternativa 2, por sua vez gera um esquema com mais coleções, sendo indicada para um *workload* que acessa principalmente as instâncias das entidades especializadas ou quando a maioria dos atributos e relacionamentos pertencem à elas. Ela também não está livre de restrições de integridade caso a especialização seja disjunta. Por fim, a alternativa 3 se mostra interessante quando qualquer instância do nível hierárquico é relevante para o *workload*.

[2]Depende do tipo de especialização definida para o nível hierárquico no esquema EER: total/parcial e disjunta/compartilhada.

A segunda subetapa analisa os relacionamentos do esquema EER, através de uma ordenação baseada nas suas cardinalidades máximas (prioridade para relacionamentos 1-1 seguido de relacionamentos 1-n e, por fim, relacionamentos m-n), e aplica uma dentre as seguintes regras de conversão:

1. *fusão*: gera-se uma coleção que une os atributos do relacionamento e das entidades envolvidas nele. Um exemplo é mostrado na Figura 2.9 (a);
2. *aninhamento*: gera-se uma coleção C_x que representa uma das entidades envolvidas no relacionamento. A outra entidade participante é representada como um bloco aninhado à C_x. Um exemplo é mostrado na Figura 2.9 (b);
3. *referência*: gera-se uma coleção para cada uma das entidades participantes do relacionamento. Pelo menos uma destas entidades abriga o relacionamento, que pode ser representado apenas como um atributo de referência (relacionamento sem atributos) ou um bloco aninhado (relacionamento com atributos). Um exemplo é mostrado na Figura 2.9 (c).

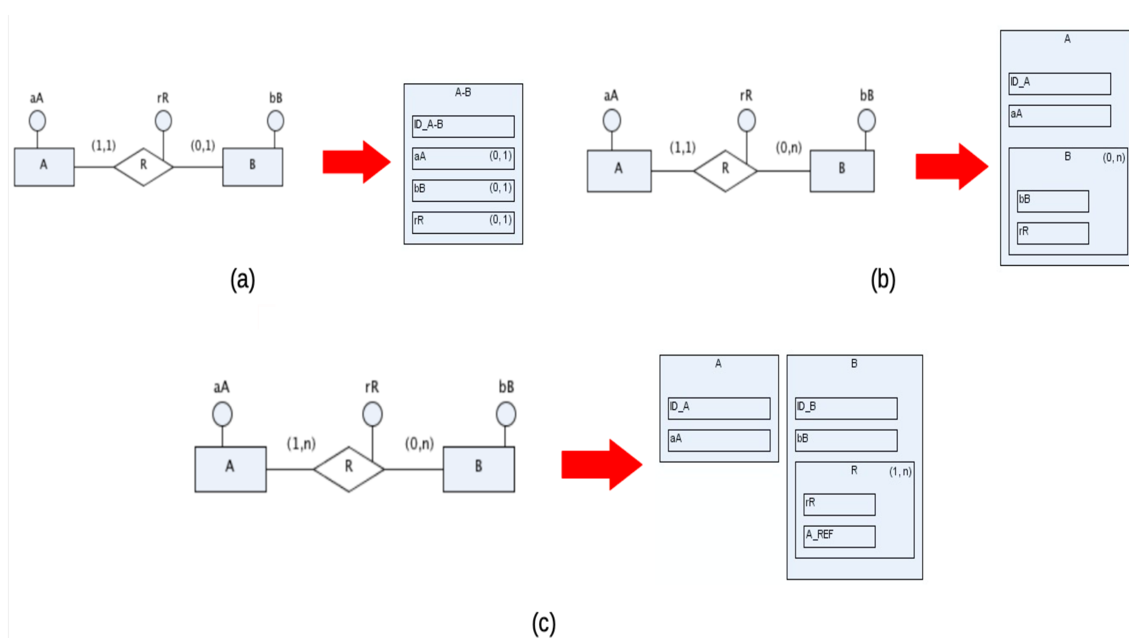


Figura 2.9. Conversão de relacionamentos em um esquema EER

A Tabela 2.1 sumariza as recomendações de aplicação destas alternativas de conversão conforme a cardinalidade do relacionamento. A alternativa *fusão* é indicada para relacionamentos 1-1, em particular relacionamentos obrigatórios, pois as instâncias estão sempre conectadas. Para casos 1-1 com opcionalidades envolvidas pode-se adotar outras alternativas, pois a fusão pode ser uma estratégia inadequada se poucas instâncias das entidades envolvidas participam do relacionamento no mundo real.

Relacionamentos 1-n podem ser convertidos pelas alternativas de *aninhamento* e *referência*. A primeira é sugerida quando uma entidade E_x se relaciona com cardinalidade

Tabela 2.1. Conversão de relacionamentos de um esquema EER baseada na cardinalidade

cardinalidade	fusão	aninhamento	referência
(1,1)-(1,1)	X		
(1,1)-(0,1)	X	X	
(0,1)-(0,1)	X	X	X
(1,1)-(1,n) / (1,1)-(0,n)		X	X
(0,1)-(0,n)		X	X
(1,n)-(1,n) / (1,n)-(0,n) / (0,n)-(0,n)			X

máxima 1 de forma obrigatória, denotando que E_x está sempre atrelada a uma única entidade E_y e pode, assim, estar aninhada à coleção que representa E_y. A segunda alternativa é mais indicada quando há opcionalidades em ambos os casos e poucas instâncias de E_x participam do relacionamento no mundo real.

Por fim, relacionamentos m-n são convertidos através da alternativa de *referência* e, neste caso, sugere-se que o bloco aninhado contendo o relacionamento seja posicionado na coleção correspondente à entidade E_i que ou possui maior probabilidade de participar do relacionamento ou E_i é a entidade mais solicitada no *workload* da aplicação para fins de navegação pelo relacionamento. Cabe observar aqui que, diferente da alternativa de conversão para BDs relacionais, relacionamentos m-n não geram uma coleção própria. Essa alternativa não é considerada aqui para evitar a geração de mais uma coleção no BD NoSQL e também por que o modelo de agregados permite definir blocos aninhados a coleções com cardinalidade associada, o que é capaz de simular um relacionamento m-n em uma direção e sentido específicos. Nada impede que o projetista do BD defina blocos aninhados contendo o relacionamento em ambas as coleções geradas para as entidades envolvidas caso esse cenário seja mais favorável para o *workload* da aplicação. Este raciocínio de conversão pode ser aplicado também a relacionamentos n-ários, ou seja, relacionamentos ternários ou superiores. Nestes casos, blocos com referências para as coleções correspondentes às entidades envolvidas são definidos.

2.3.4. Projeto Lógico de Baixo Nível para BDs NoSQL

Esta subetapa é responsável pela conversão do esquema de agregados para esquemas representados nos modelos chave-valor, orientados a colunas e/ou orientados a documentos. Estes 3 modelos de dados NoSQL são centrados em acesso via chave e podem suportar conteúdos com estruturas aninhadas, apresentando uma forte similaridade com o modelo de agregados. Nesta etapa, cada coleção do esquema de agregados é analisada e convertida em uma estrutura adequada a um modelo de dados NoSQL.

A Tabela 2.2 sumariza o mapeamento de uma coleção para o modelo chave-valor. Este mapeamento é mais simples em função da simplicidade do modelo chave-valor. Neste caso, o ID da coleção se torna a chave de um esquema chave-valor e o conteúdo completo da coleção deve ser desaninhado e serializado gerando um conjunto de pares chave-valor para ser inserido no componente valor.

A Figura 2.10 mostra um exemplo deste mapeamento considerando novamente o

Tabela 2.2. Conversão de agregados para chave-valor

modelo de agregados	modelo chave-valor
ID da coleção	chave
conteúdo da coleção	valor

domínio de um Museu. Supondo o esquema no modelo de agregados na parte esquerda da figura e uma instância de *Obra* neste esquema com identificação *O145* referente à pintura *Abaporu* de *Tarsila do Amaral*, ela se torna uma instância em um BD chave-valor, conforme mostrado à direita da figura. Blocos internos são desaninhados e serializados, como é o caso de *autoria*. Conteúdos multivalorados, como o atributo *horários* em *Salão*, são igualmente serializados.

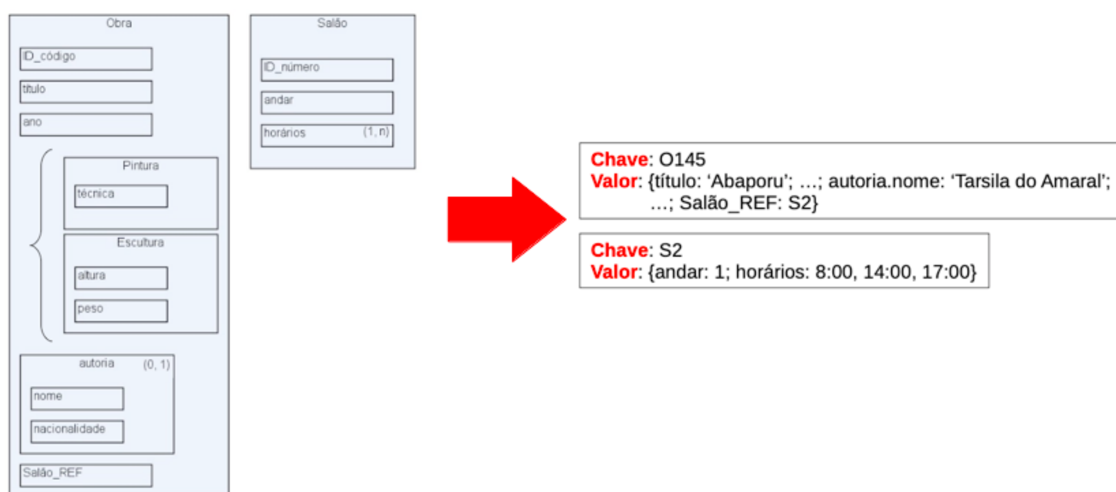


Figura 2.10. Exemplo de conversão de um esquema de agregados para um esquema chave-valor

A Tabela 2.3 sumariza o mapeamento de uma coleção para o modelo orientado a colunas. Este modelo de BD NoSQL, por ser mais robusto, permite um mapeamento mais detalhado. Entretanto, devido à falta de homogeneidade nas implementações deste modelo nos vários SGBDs orientados a colunas, algumas recomendações de mapeamento podem ser implementadas de formas diferentes, como são os casos do mapeamento de atributo multivalorado e bloco. No primeiro caso, se o SGBD suportar colunas multivaloradas, o mapeamento é trivial. Senão, o conteúdo do atributo deve ser serializado. No segundo caso, se o SGBD suportar supercolunas, o mapeamento é igualmente trivial. Caso contrário, o conteúdo do bloco deve ser desaninhado e serializado.

Um exemplo de conversão é mostrado na Figura 2.11, supondo o suporte à colunas multivaloradas e supercolunas. As instâncias exemplo são as mesmas mencionadas anteriormente. Basicamente, cada coleção se torna uma família de colunas.

Por fim, a Tabela 2.4 sumariza o mapeamento de uma coleção para o modelo orientado a documentos. Este modelo de BD NoSQL é o que apresenta maior afinidade com o modelo de agregados.

Um exemplo de conversão é mostrado na Figura 2.12 considerando que o SGBD

Tabela 2.3. Conversão de agregados para orientado a colunas

modelo de agregados	modelo orientado a colunas
coleção	família de colunas
ID da coleção	chave da família de colunas
atributo simples	coluna
atributo de referência	coluna
atributo multivalorado	coluna multivalorada ou coluna com conteúdo serializado
bloco	supercoluna ou conteúdo desaninhado e serializado

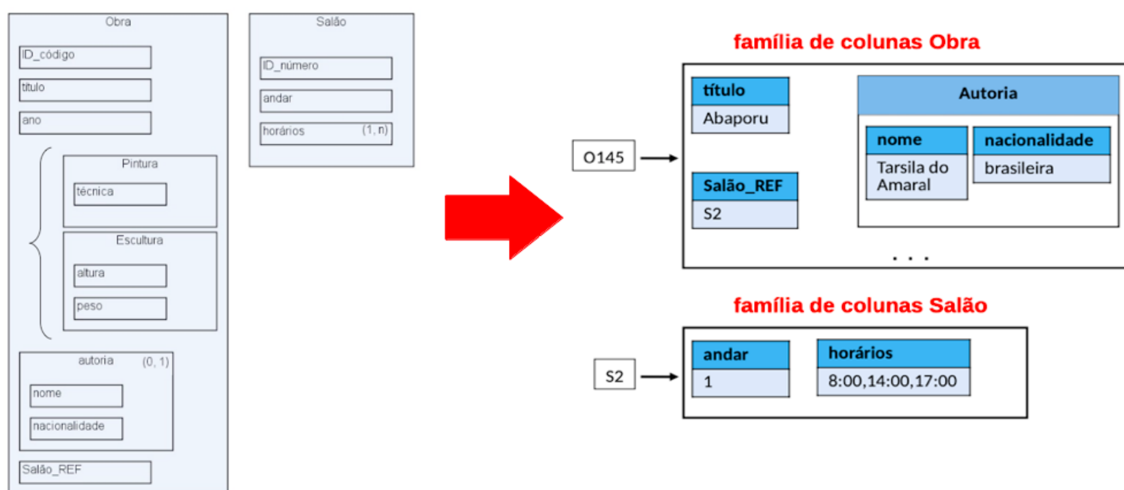


Figura 2.11. Exemplo de conversão de um esquema de agregados para um esquema orientado a colunas

Tabela 2.4. Conversão de agregados para orientado a documentos

modelo de agregados	modelo orientado a documentos
coleção	documento
ID da coleção	chave do documento
atributo simples	atributo simples
atributo de referência	atributo simples
atributo multivalorado	atributo do tipo lista
bloco	atributo do tipo objeto

orientado a documentos segue o formato JSON para fins de armazenamento. Os recursos do modelo de dados JSON, como atributos do tipo lista e do tipo objeto, permitem uma conversão bastante direta de uma instância complexa do modelo de agregados para um objeto JSON.

Na sequência aborda-se o projeto lógico para BDs orientados a grafos.

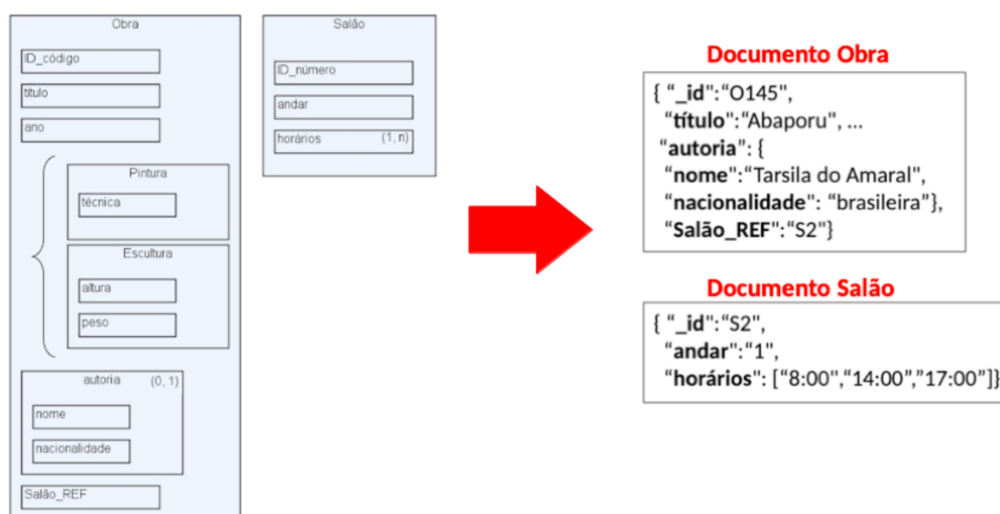


Figura 2.12. Exemplo de conversão de um esquema de agregados para um esquema orientado a documentos

2.3.5. Projeto Lógico para BDs NoSQL Orientado a Grafos

O modelo de dados orientado a grafos se diferencia dos demais modelos de dados NoSQL pelo foco das suas consultas, que são centradas em relacionamentos ao invés de acessos via chave. Em geral, seus atributos possuem domínios simples, com alguns SGBDs oferecendo suporte a atributos do tipo *array*. Desta forma, ao invés de considerar um mapeamento intermediário para o modelo de agregados, sugere-se um mapeamento direto de um esquema EER para um esquema orientado a grafos, considerando que um esquema EER já é uma estrutura em forma de grafo.

A Tabela 2.5 apresenta o mapeamento de um esquema EER para um esquema orientado a grafo. Basicamente, uma entidade se torna um tipo de vértice e um relacionamento binário um tipo de aresta, sendo que ambos podem conter propriedades, e os nomes de entidades e de relacionamentos binários se tornam os rótulos dos vértices e das arestas, respectivamente.

Tabela 2.5. Conversão de um esquema EER para um BD orientado a grafos

modelo EER	**modelo orientado a grafos**
entidade	vértice
atributo simples	propriedade
atributo composto	vértice
atributo multivalorado	propriedade *array* ou vértice
relacionamento binário	aresta
relacionamento n-ário	vértice

Arestas em BDs orientados a grafos geralmente são unidirecionais. Logo, a decisão pelos vértices de origem e de destino podem ser guiados pelo *workload* típico da aplicação, ou seja, qual sentido é mais frequente nas buscas. Caso ambos os sentidos sejam frequentemente percorridos em buscas, deve-se criar duas arestas com sentidos opostos entre os vértices em questão.

Com relação aos atributos do modelo EER, atributos simples tornam-se propriedades de vértices ou de arestas. Atributos compostos, não suportados em geral pelos SGBDs orientados a grafos, devem gerar novos vértices no grafo conectados ao vértice que representa a entidade com o atributo composto. Já atributos multivalorados podem ser convertios em propriedades do tipo *array*, caso o SGBD orientado a grafo o suporte, ou devem gerar vértices para representar cada valor, além de arestas que os conectem ao vértice que representa a entidade.

Por fim, sugere-se que um relacionamento n-ário R_n seja mapeado para um vértice no grafo. Isso evita a manutenção de muitas referências em cada vértice representativo de cada entidade E_i participante de R_n. Cada E_i deve ter arestas para as ocorrências de R_n que participa.

A literatura carece de uma metodologia efetiva para o projeto lógico de um BD orientado a grafo. A metodologia aqui proposta se baseia nas recomendações do recente trabalho de Silva & Mello [Silva and Mello 2021] e nas regras de conversão presentes na Tabela 2.5. O processo de conversão é composto de 3 etapas: *(i) conversão de entidades; (ii) conversão de hierarquias*; e *(iii) conversão de relacionamentos*. A primeira etapa analisa entidades não envolvidas em hierarquias de especialização, gerando um tipo de vértice para cada uma delas. A segunda etapa analisa hierarquias de especialização de entidades (percorrimento *bottom-up*), sendo possível fusionar a entidade genérica com uma ou mais entidades especializadas em um único tipo de vértice. Por fim, os relacionamentos são analisados e convertidos em tipos de aresta com uma cardinalidade associada. Também é possível fusionar entidades conectadas por relacionamentos 1-1 em um único tipo de vértice para reduzir o tamanho do esquema, caso grande parte das instâncias destas entidades participam do relacionamento.

A Figura 2.13 ilustra um exemplo de conversão. O esquema EER mostrado na Figura 2.13 (a) é um esquema simplificado para o domínio de um Museu. Um possível esquema lógico orientado a grafo é mostrado na Figura 2.13 (b). As entidades *Salão*, *Autor* e *TiposPedra* são tratadas pela primeira etapa gerando os respectivos tipos de vértices.

Na sequência, a hierarquia de especialização é analisada iniciando pelo nível de *Escultura* e suas especializações. Neste caso, optou por fusionar esse nível em um único tipo de vértice (Escultura) considerando que a especialização é compartilhada e para redução do tamanho do esquema. O nível superior é então tratado, gerando um tipo de vértice para cada entidade (*Obra*, *Pintura* e *Escultura*) e arestas conectando-os.

Por fim, analisa-se os relacionamentos. Optou-se pela fusão da entidade *Zelador* com a entidade *Salão* pelo fato do relacionamento ser do tipo 1-1 obrigatório. Os demais relacionamentos podem ser convertidos considerando o *workload* da aplicação para determinar a melhor direção e sentido da aresta. No caso do relacionamento entre *Salão* e *Obra* optou-se por gerar 2 arestas com sentidos diferentes supondo que este relacionamento é percorrido com frequência em ambos os sentidos.

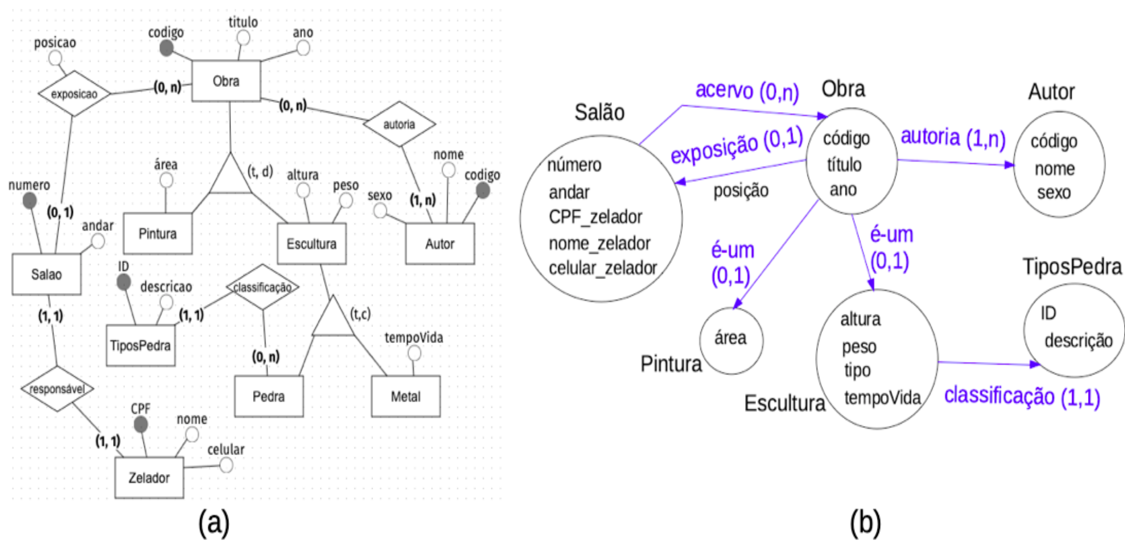


Figura 2.13. Exemplo de conversão de um esquema EER para um esquema orientado a grafos

2.4. Implementação de um Projeto lógico de BD NoSQL

A implementação do projeto lógico é a etapa que realiza a transformação do projeto lógico em um projeto físico específico de um SGBD. Enquanto na modelagem conceitual o foco é esboçar a relação entre os conceitos do domínio, o modelo lógico foca no armazenamento considerando as peculiaridades de cada modelo de dados. O modelo físico de um BD é ainda mais específico considerando as características do SGBD que será utilizado para o armazenamento dos dados. Desta forma, nesta etapa, são realizadas escolhas que definem como se dará o acesso aos dados e como otimizar esse processo.

O projeto de um BD relacional permite que o usuário seja capaz de combinar diferentes tabelas com junções para gerar o resultado de uma consulta. O projeto de um BD NoSQL, entretanto, não permite, de maneira geral, junções de diferentes estruturas de dados. Esta limitação faz com que os dados sejam organizados de tal maneira que uma junção, por exemplo, não seja necessária.

Durante a implementação do esquema lógico em um SGBD NoSQL é necessário que se leve em consideração o *workload* das principais consultas. As consultas mais comuns devem ser utilizadas como guia na transformação do esquema lógico. Durante a transformação considera-se constantemente o *tradeoff* do custo efetivo de consultar os dados que estão armazenados através de agregados ou a necessidade de combiná-los em memória através de operações realizadas pela aplicação. Desta forma, é importante manter em mente que o esquema deve atender de maneira efetiva as consultas mais frequentes.

Outro ponto importante a ser considerado na implementação do esquema lógico é o SGBD NoSQL a ser utilizado. Essa é uma decisão complexa que envolve um estudo das características básicas da aplicação. A escolha de uma modelo de dados inadequado para um determinado problema, ou mesmo, um SGBD inadequado pode levar a solução proposta a problemas de escalabilidade no geral. Cada modelo de dados e cada SGBD atendem um conjunto de requisitos e um conjunto de aplicações distinto. Por exemplo, se a modelagem dos dados em questão possui alto nível de relacionamentos, é provável que um SGBD orientado a grafo seja uma boa opção. Ou mesmo, se a aplicação exige um grande número de buscas por atributos e os dados não tem muitos relacionamentos, um SGBD orientado a grafos pode não ser uma boa opção, sendo um SGBD orientado a documentos mais adequado. A escolha do modelo e do SGBD é muito dependente dos requisitos, e é importante que essa escolha seja realizada de maneira consciente para que sejam evitados possíveis problemas posteriores.

Nas próximas seções são discutidas duas implementações de esquemas lógicos nos SGBDs NoSQL MongoDB e Neo4J por serem os mais populares no mundo NoSQL. Como base é utilizado o esquema EER para um domínio de uma rede social apresentado na Figura 2.14. O domínio é uma rede social simples onde um *Perfil* possui pode postar, comentar e curtir (*like*) um *Post*. Adicionalmente, para cada *perfil* são armazenadas informações pessoais (*Sobre*). Para cada um dos SGBDs NoSQL apresentados é proposto um mapeamento lógico e físico para o domínio em questão.

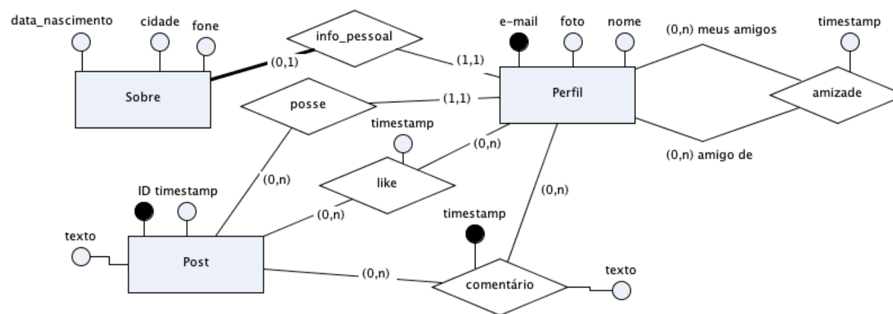


Figura 2.14. Exemplo de esquema EER para uma rede social

2.4.1. Implementação no SGBD MongoDB

O MongoDB é um SGBD NoSQL cujo modelo de dados é orientado a documentos. Ele provê alta disponibilidade, escalabilidade e uma forma flexível de armazenar os dados. Segundo o *ranking* do *DB-Engine*[3], o Mongo DB é o quinto SGBD mais utilizado na atualidade. Sua estrutura de dados é baseada no formato *JSON*, armazenando seus dados em documentos *BSON* (*JSON* binário).

A estrutura interna de um BD em *MongoDB* pode ser definida através de um conjunto de coleções de documentos. Cada coleção armazena um conjunto de documentos que, em geral, possui uma similaridade estrutural ou semântica. Analogamente ao modelo relacional, uma coleção de documentos pode ser vista como uma tabela do BD relacional [Sadalage and Fowler 2013]. Por sua vez, cada documento é composto por um conjunto de atributos e valores, análogo a uma tupla.

Ao adotar o *MongoDB* alguns comportamentos internos do SGBD devem ser considerados. Diferente de um BD relacional tradicional, o *MongoDB* não suporta transações ACID. Para este SGBD as propriedades ACID são consideradas apenas em nível de documento, ou seja, durante a inserção de um documento ou ele insere o documento por inteiro ou não insere nada. Quando a inserção envolve mais de um documento, cada documento é tratado como um elemento independente.

Outro ponto importante a ser considerado é a estrutura de armazenamento. O *MongoDB* possui uma arquitetura distribuída do tipo *master* e *worker*. Qualquer nodo pode tratar demandas de leitura, mas todas as escritas são executadas no *master* para depois serem replicadas para os nodos *workers*. Desta forma, a consistência aplicada, por padrão, é eventual tendo em vista que se um dado *x* foi escrito no nodo mestre e ainda não foi replicado para um *worker* e este atender uma requisição de leitura de *x*, ele responderá com os dados desatualizados. É importante ressaltar que o tempo de replicação dos dados entre os nodos tem uma latência baixa, mas mesmo assim, não é desprezível para alguns domínios de aplicação.

O MongoDB também suporta níveis maiores de consistência usando uma parâmetro de escrita denominado de *WriteConcern*[4]. O parâmetro define o número de réplicas que deve receber o dado de maneira síncrona. Por exemplo, em um *cluster* composto de

[3]`https://db-engines.com/en/ranking`
[4]`https://www.mongodb.com/docs/manual/reference/write-concern/`

3 nodos (1 *master* e 2 *workes*) se o *write concern* for 2, o MongoDB irá gravar os dados no *master* e em pelo menos um *worker* e só então considerar que a operação foi realizada. Aumentando o *write concern* é possível atingir um alto nível de consistência, porém, a latência de atualização entre as réplicas pode vir a ser um gargalo.

Para a implementação do esquema lógico no MongoDB todas as características supracitadas devem ser consideradas. A escolha de um SGBD para o armazenamento dos dados vem recheada de prós e contras e é importante que o projetista esteja ciente delas para realizar a escolha correta. Além disso, o usuário deve conhecer o *workload* da aplicação pois a modelagem física em um sistema NoSQL é muito pautada nas consultas a serem executadas.

Não é possível criar um esquema físico para o MongoDB que possa responder com alto desempenho qualquer consulta. Um dos motivos disso é a falta de algoritmos de junção, fazendo com que para unir dados de diferentes coleções de documentos sejam tratados pela própria aplicação. Desta forma, ao modelar deve-se levar em consideração quais são as consultas com maior custo e quais são mais executadas e armazenar os dados de maneira que estas tenham o melhor desempenho possível.

Considerando o exemplo do domínio da rede social (Figura 2.14) um possível esquema lógico é apresentada na Figura 2.15. Ele considera que é comum, por exemplo, consultas que por um *post* e seus comentários bem como o número de *likes* de um *post*. Porém, na estrutura proposta, torna-se mais complexo realizar consultas como buscar todos os *posts* de um determinado usuário ou o nome do usuário que realizou um comentário.

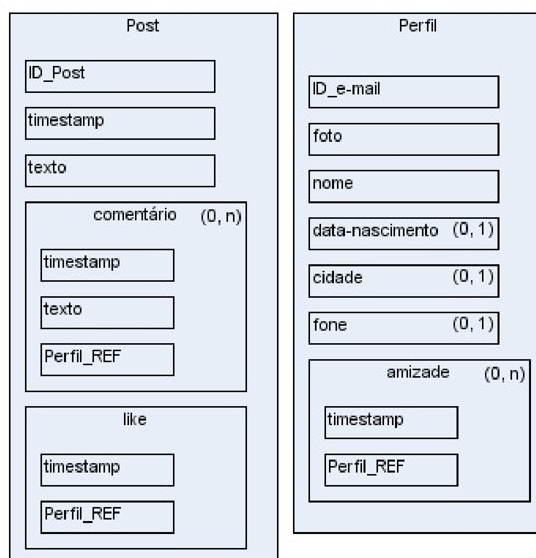


Figura 2.15. Exemplo de modelo lógico baseado em agregado.

É possível ver Figura 2.15 na que cada *Post* possui um id, um texto e conjuntos de comentários e *likes*. Cada comentário possui um texto e a referência do perfil que o criou. No Mongo DB não existe o conceito de chave estrangeira. Assim, esse atributo refere-se unicamente ao código de um documento visto com um valor inteiro qualquer.

O SGBD não realiza nenhuma verificação de integridade referencial. Apenas o usuário compreende a semântica desta ligação.

Este armazenamento obriga o usuário a implementar um algoritmo de junção para exibir o nome de um usuário que efetuou um comentário em um *post* específico. Este processo é oneroso e deve ser considerado durante a etapa de modelagem. Se esta é uma consulta comum ao sistema uma forma de mitigar a junção seria junto ao comentário já armazenar o nome do usuário que o fez. Isto aumenta a redundância de dados mas evita gasto com processamento de uma junção. Este tipo de *trade-off* deve ser considerado constantemente no processo de modelagem para que o melhor modelo possível seja gerado. Um bom esquema deve ser abrangente o suficiente para suportar uma grande gama de consultas, porém otimizado para as consultas mais comuns.

A linguagem de consulta do MongoDB se difere muito do padrão SQL. Como o SGBD trabalha com o formato JSON, seu *shell* de consulta é baseado na linguagem *Javascript*. Neste minicurso não são abordadas a instalação e configuração para o MongoDB. São apresentados alguns comandos básicos da sua *Application Programming Interface (API)* de acesso para inserção de dados e posterior consulta.

Para inserir dados no MongoDB a sua API oference os comandos *insertOne* e *insertMany*. O primeiro recebe como parâmetro um único documento a ser inserido. Já o segundo recebe um vetor com vários documentos a serem inseridos. A Figura 2.16 apresenta um exemplo de inserção de dados, sendo que $db.perfil$ representa o BD atual *db* e a coleção de documentos *perfil* onde o documento Json é inserido.

```
1 db.perfil.insertOne({
2   "_id": 2,
3   "nome": "Violet Baudelaire",
4   "data_nascimento": new Date("1994-08-01"),
5   "cidade": "Dream of Stone",
6   "amizade": [1,3,5,8,7]
7 });
```

Figura 2.16. Exemplo *insert*.

Para consulta a dados, a API do MongoDB utiliza o comando *find*. Este comando recebe dois parâmetros. O primeiro é um documento que contém a busca a ser realizada e o segundo a projeção dos dados (atributos retornados). O documento de busca geralmente define operadores a serem executados, permite o uso de conectores lógicos *and* e *or*, assim como operadores lógicos > ($\$gt$), < ($\$lt$), >= ($\$gte$), <= ($\$lte$) e diferente ($\$ne$).

A Figura 2.17 apresenta um exemplo de uma busca de dados no MongoDB. Ela lista perfis ($db.perfil.find$) que sejam válidos para o predicado informado. O predicado considera um operador lógico *and* representado pelo vetor de todos os filtros conectados. Cada filtro deve explicitar sobre qual atributo deve ser executado, qual o tipo de comparação a ser feita e qual o valor usado na comparação. No caso do exemplo, a consulta retorna todos os perfis cujo nome seja *Violet Baudelaire* e tenha data de nascimento superior a $01/01/2000$.

Apesar da diferença em relação à SQL, o comando de consulta da API do MongoDB possui um alto poder de expressão, permitindo a definição de diversas consultas

```
1 db.perfil.find({
2   $and: [
3     {"nome": "Violet Baudelaire"},
4     {"data_nascimento": {
5         $gte: new Date("2000-01-01")
6       }
7     }]}, {}
8 });
```

Figura 2.17. Exemplo *busca*.

complexas, com a ressalva que consultas com junções não são suportadas, como mencionado anteriormente. Uma consulta particularmente complexa para ser resolvida é listar o nome de todos os amigos de um determinado perfil. Para defini-la, deve-se buscar as informações de cada *id* de amigo armazenado no vetor e realizar a junção com a própria coleção de documentos *perfil*. Se expandirmos a consulta para mostrar o nome dos amigos dos amigos de um dado perfil então a consulta fica inviável. Por outro lado, esse tipo de consulta é plenamente viável em um BD orientado a grafo, como o Neo4j, cuja implementação é detalhada a seguir.

2.4.2. Implementação no SGBD Neo4j

O Neo4J é um SGBD orientado a grafos. Sua estrutura interna é otimizada para o armazenamento de dados que possuam um alto número de relacionamentos. Diferente do MongoDB, que pode ser adaptado e aplicado em diversas gamas de problemas, o Neo4J é otimizado para o armazenamento de dados que possuam muitos relacionamentos.

O modelo de dados do Neo4J é baseado majoritariamente em dois conceitos: *vértices* e *arestas*. Um vértice é definido por um *label*, um ou mais tipos e um conjunto de atributos armazenado no formato JSON. Já uma aresta é um relacionamento que conecta dois vértices. Cada aresta possui um vértice de origem, um vértice de destino, um *label*, um ou mais tipos e um conjunto de atributos também no estilo JSON.

Diferente da maioria dos BDs NoSQL, o Neo4J possui suporte às propriedades ACID. Quando um conjunto de dados é adicionado ou atualizado em conjunto, todas as operações devem ser executadas até o fim, ou nenhuma delas é efetivada. Porém, para que o ACID seja melhor controlado, o Neo4J não permite o particionamento dos dados.

O Neo4J utiliza a linguagem *Cypher* para executar suas operações. Cypher é uma linguagem declarativa imperativa com alto poder de expressão e que possui semelhanças com a SQL. Considerando o domínio da rede social (Figura 2.14), um possível esquema lógico é apresentado na Figura 2.18. Diferente do esquema orientado a documentos apresentado anteriormente, o modelo de BD orientado a grafo é focado nos relacionamento dos dados. Desta forma, os relacionamentos *postar*, *comentar* e dar *like* são mapeados diretamente para arestas que unem vértices do tipo *Perfil* com *Post*. Além disso, o autorelacionamento de um perfil (*amizade*) também é mapeado para uma aresta.

O modelo de dados orientado a grafo permite que o projetista defina um conjunto de atributos para cada vértice e aresta. O Neo4j permite que sejam definidos filtros por valores de atributos, entretanto, este tipo de consulta não é tão otimizado quanto buscas

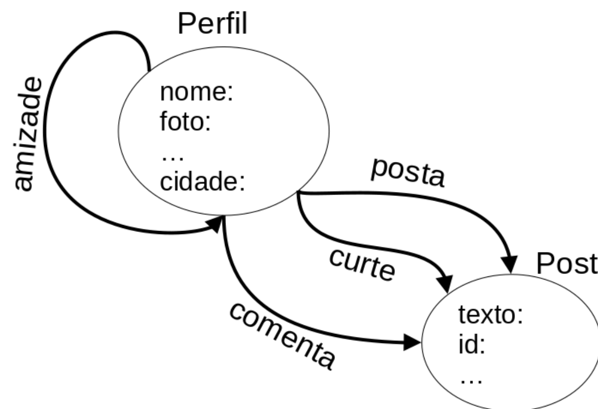


Figura 2.18. Exemplo de esquema lógico para um BD em grafo

envolvendo navegação pelas arestas.

A inserção de dados no Neo4J é realizada através do comando *CREATE* da linguagem Cypher. Para tanto, elementos delimitados por ( ) são considerados vértices e elementos delimitados por $-[]->$ são arestas. Existem duas formas de realizar a inserção dos dados. A primeira realiza a criação de todos os elementos diretamente em um único comando. Um exemplo é apresentado nas 2 primeiras linhas da Figura 2.19. O comando cria dois vértices (*Violet* e *Sunny*), ambos do tipo *Perfil*. O tipo serve para indicar vértices que possuem a mesma estrutura e semântica.

```
CREATE  (s:Perfil {name: "Sunny"}),
        (v:Perfil {name: "Violet"});


MATCH   (s:Perfil {name: "Sunny"}),
        (v:Perfil {name: "Violet"})
CREATE (s)-[:rel]->(v)
```

Figura 2.19. Exemplos de inserção de dados na linguagem Cypher

Já a segunda alternativa utiliza o comando *MATCH* além do comando CREATE. O comando MATCH é utilizado para buscar elementos em um grafo. No exemplo da Figura 2.19, busca-se 2 vértices do tipo *Perfil*, filtrados pelo atributo *nome*, e define-se uma aresta entre os dois vértices, caso eles sejam encontrados.

A tipagem (*labels*) dos vértices e das arestas em um grafo é um importante recurso na definição de consultas. Cada vértice ou aresta pode possuir mais de um tipo, o que permite uma maior flexibilidade de representação. A Figura 2.20 apresenta uma representação gráfica de um conjunto de dados armazenado no Neo4J. Vértices de mesma cor são de um mesmo tipo. Perceba também que as arestas são todas unidirecionais. O Neo4J não suporta arestas bidirecionais.

Conforme descrito anteriormente, a consulta a dados na linguagem Cypher é definida pelo comando *MATCH*. Este comando visa encontrar um conjunto (mesmo que unitário) de vértices e arestas que satisfaçam suas restrições. Para cada vértice ou aresta

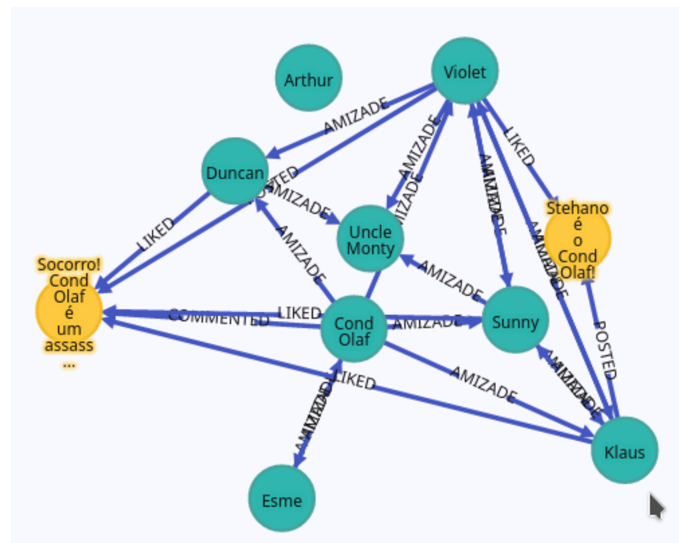


Figura 2.20. Exemplo de uma instância de grafo no Neo4j

envolvido na consulta é possível atribuir um *alias* facilitando operações complexas que exijam filtros com operadores *and* e *or*.

```
MATCH (s:Perfil)-[:AMIZADE]->(y)
WHERE s.nome = 'Sunny'
RETURN y.nome, y.data_nascimento

MATCH (s:Perfil)-[:AMIZADE]->()-[:AMIZADE]->(y)
WHERE s.nome = 'Sunny'
RETURN y.nome
```

Figura 2.21. Exemplos de consultas na linguagem Cypher

A Figura 2.21 apresenta 2 consultas em Cypher. A primeira retorna o nome e a data de nascimento de todos os vértices que possuem um relacionamento do tipo amizade com o vértice de nome *Sunny*. Apesar de uma consulta relativamente simples, este exemplo ilustra a estrutura básica de uma consulta na linguagem Cypher. A primeira parte da consulta é destinada a busca por um padrão (*MATCH*), a segunda parte é o predicado da consulta (*WHERE*) e, por fim, o resultado da consulta (*RETURN*).

Já a segunda consulta é um pouco mais complexa. Ela realiza a seleção dos vértices (*y*) que possuem um relacionamento de amizade com algum nodo que já possui relacionamento de amizade com o vértice *Sunny*. Em outras palavras, a consulta retorna o nome dos amigos dos amigos de *Sunny*. Esta é uma consulta que teria um alto custo de execução em outros modelos de dados NoSQL, entretanto, no modelo orientando a grafos é relativamente simples devido à sua ênfase no percorrimento por relacionamentos entre dados.

2.5. Considerações Finais

Este minicurso apresenta uma proposta de metodologia para projeto lógico de BDs NoSQL a partir de um esquema conceitual EER. Esta metodologia contempla todos os 4 modelos de dados NoSQL e também discute como a implementação de um esquema lógico para NoSQL pode ser implementada em 2 SGBDs de amplo uso: MongoDB (orientado a documentos) e Neo4j (orientado a grafos).

Esta metodologia é uma contribuição para a área de modelagem de dados NoSQL, sendo um guia detalhado do processo de mapeamento de esquemas conceituais para projetistas que desejam utilizar SGBDs NoSQL para o armazenamento e acesso aos seus dados. A literatura de BD é carente de tal metodologia detalhada. A proposta aqui apresentada se baseia no projeto de BD clássico e na adaptação das regras de conversão EER-relacional. Como em qualquer projeto de BD para um dado modelo lógico e físico de dados, a metodologia proposta apresenta diversas alternativas de conversão de conceitos de um esquema EER para estruturas correspondentes nos modelos de dados NoSQL, cabendo ao projetista decidir pelas alternativas que melhor contemplam as necessidades de armazenamento e acesso aos seus dados.

O *Grupo de BD da UFSC*[5] vêm trabalhando em soluções informatizadas para o projeto de BDs, com destaque para o protótipo *brModeloNext*, que está sendo estendido para suportar a modelagem lógica de agregados e consequente geração de esquemas físicos em alguns SGBDs NoSQL. Espera-se que em breve esta ferramenta esteja estável e disponível para uso pela comunidade acadêmica para ensino e prática de projeto de BD NoSQL, bem como para demais interessados.

[5] *http://lisa.inf.ufsc.br/; https://github.com/gbd-ufsc*

Referências

[Atzeni et al. 2014] Atzeni, P., Bugiotti, F., and Rossi, L. (2014). Uniform Access to NoSQL Systems. *Information Systems*, 43(SI):117–133.

[Elmasri and Navathe 2011] Elmasri, R. and Navathe, S. (2011). *Fundamentals of Database Systems*. Addison-Wesley, 6 edition.

[Frozza and Mello 2020] Frozza, A. and Mello, R. (2020). JS4Geo: A Canonical JSON Schema for Geographic Data Suitable to NoSQL Databases. *GeoInf.*, 24(4):987–1019.

[Henry Korth and Silberschatz 2019] Henry Korth, S. S. and Silberschatz, A. (2019). *Database System Concepts*. McGraw-Hill Education, 7 edition.

[Heuser 2009] Heuser, C. A. (2009). *Projeto de Banco de Dados*. Bookman, 6 edition.

[Lima 2016] Lima, C. (2016). Projeto Lógico de Bancos de Dados NoSQL Documento a Partir de Esquemas Conceituais Entidade-Relacionamento Estendido (EER). Master's thesis, PPGCC-UFSC.

[Lima and Mello 2015] Lima, C. and Mello, R. (2015). A Workload-driven Logical Design Approach for NoSQL Document Databases. In *XVII ACM iiWAS*, pages 73:1–73:10.

[Lima and Mello 2016] Lima, C. and Mello, R. (2016). On Proposing and Evaluating a NoSQL Document Database Logical Design Approach. *Int. J. Web Inf. Syst.*, 12(4):398–417.

[Sadalage and Fowler 2013] Sadalage, P. J. and Fowler, M. (2013). *NoSQL Distilled : A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley.

[Santana and Mello 2020] Santana, L. H. Z. and Mello, R. (2020). An Analysis of Mapping Strategies for Storing RDF Data into NoSQL Databases. In *XXXV ACM SAC*, pages 386–392.

[Schreiner et al. 2020] Schreiner, G., Duarte, D., and Mello, R. (2020). Bringing SQL Databases to Key-based NoSQL Databases: A Canonical Approach. *Computing*, 102(1):221–246.

[Schreiner et al. 2015] Schreiner, G. A., Duarte, D., and Mello, R. (2015). SQLtoKeyNoSQL: A Layer for Relational to Key-based NoSQL Database Mapping. In *XVII ACM iiWAS*, pages 74:1–74:9. ACM.

[Schreiner et al. 2019] Schreiner, G. A., Duarte, D., and Mello, R. (2019). When Relational-Based Applications Go to NoSQL Databases: A Survey. *Information*, 10(7):241–263.

[Silva and Mello 2021] Silva, T. H. V. and Mello, R. (2021). A Rule-based Conversion of an EER Schema to Neo4j Schema Constraints. In *XXXVI SBBD*, pages 181–192.

[Stonebraker 2010] Stonebraker, M. (2010). SQL Databases vs. NoSQL Databases. *Commun. ACM*, 53(4):10–11.