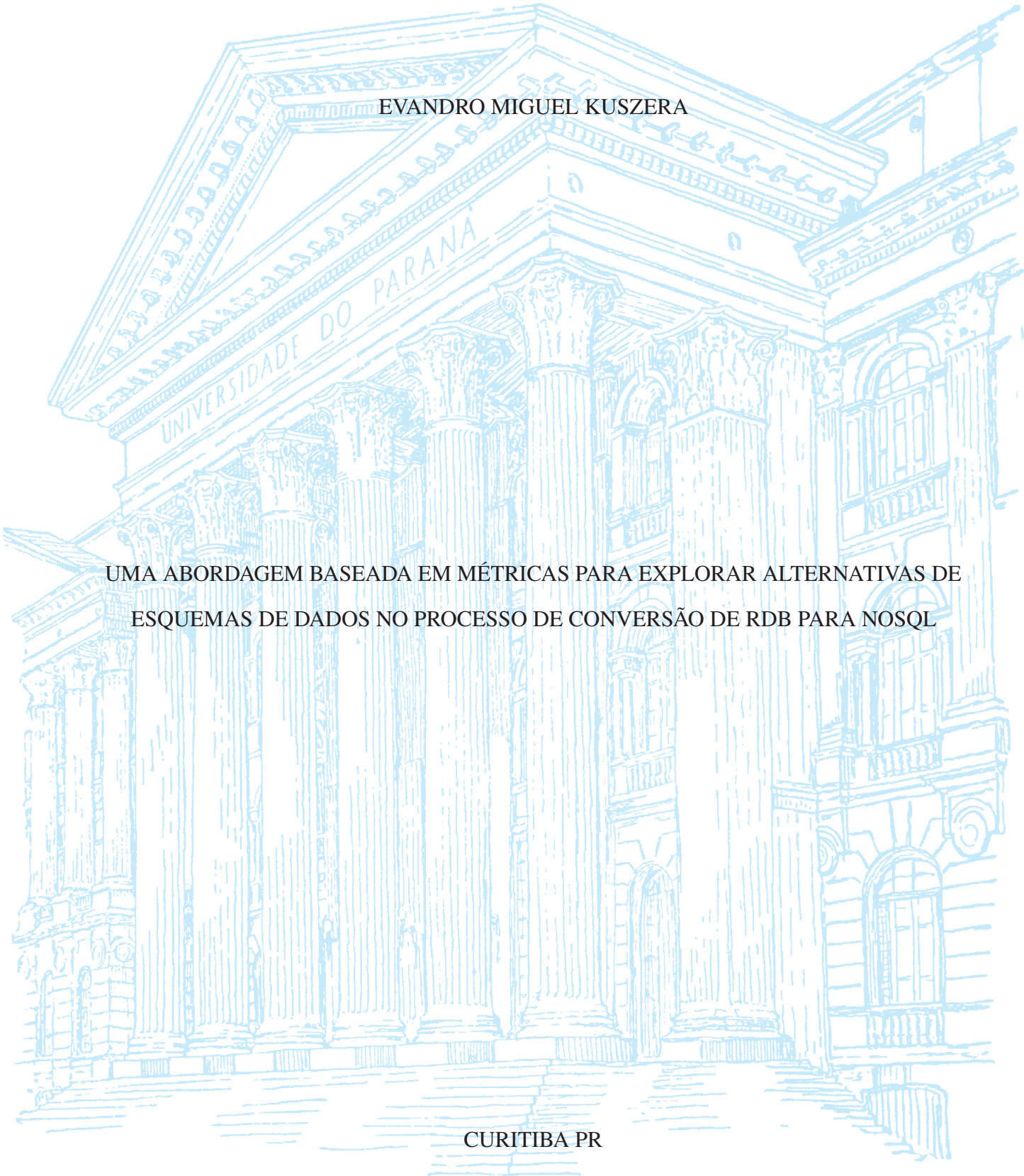
UNIVERSIDADE FEDERAL DO PARANÁ

EVANDRO MIGUEL KUSZERA



UMA ABORDAGEM BASEADA EM MÉTRICAS PARA EXPLORAR ALTERNATIVAS DE ESQUEMAS DE DADOS NO PROCESSO DE CONVERSÃO DE RDB PARA NOSQL

CURITIBA PR

2020

EVANDRO MIGUEL KUSZERA

UMA ABORDAGEM BASEADA EM MÉTRICAS PARA EXPLORAR ALTERNATIVAS DE ESQUEMAS DE DADOS NO PROCESSO DE CONVERSÃO DE RDB PARA NOSQL

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientadora: Profa. Dra. Leticia Mara Peres.

Coorientador: Prof. Dr. Marcos Didonet Del Fabro.

CURITIBA PR

2020

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

K97a Kuszera, Evandro Miguel
Uma abordagem baseada em métricas para explorar alternativas de esquemas de dados no processo de conversão de RDB para NoSQL [recurso eletrônico] / Evandro Miguel Kuszera. – Curitiba, 2020.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2020.

Orientadora: Leticia Mara Peres.
Coorientador: Marcos Didonet Del Fabro.

1. Avaliação. 2. Banco de dados relacionais. 3. Banco de dados não relacionais. I. Universidade Federal do Paraná. II. Peres, Leticia Mara. III. Del Fabro, Marcos Didonet. IV. Título.

CDD: 005.743

Bibliotecária: Vanusa Maciel CRB- 9/1928

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA - 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **EVANDRO MIGUEL KUSZERA** intitulada: **Uma Abordagem Baseada Em Métricas Para Explorar Alternativas de Esquemas de Dados no Processo de Conversão de RDB para NoSQL**, sob orientação da Profa. Dra. LETICIA MARA PERES, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 21 de Outubro de 2020.

Assinatura Eletrônica
21/10/2020 15:29:41.0
LETICIA MARA PERES
Presidente da Banca Examinadora

Assinatura Eletrônica
22/10/2020 07:26:07.0
LUIZ CELSO GOMES JUNIOR
Avaliador Externo (UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ)

Assinatura Eletrônica
04/11/2020 11:17:57.0
MARCOS SFAIR SUNYE
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
21/10/2020 16:06:39.0
EDUARDO CUNHA DE ALMEIDA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
22/10/2020 11:22:33.0
DANIEL LUCREDIO
Avaliador Externo (UNIVERSIDADE FEDERAL DE SÃO CARLOS)

Dedico à minha família, que sempre acreditou em mim e forneceu todo o suporte que eu precisava.

AGRADECIMENTOS

Agradeço à minha esposa Sheila e aos meus filhos Gabriel e Gustavo, que sempre estiveram ao meu lado, me apoiando e dando forças para continuar a caminhada. Destaco a importância da minha esposa Sheila, que muito me ajudou durante esses quatro anos, fornecendo apoio emocional e mostrando que todos temos momentos bons e ruins, que tudo na vida é aprendizado, e que quem trabalha com dedicação e persistência sempre alcança seus objetivos. Sou extremamente grato pela paciência que meus filhos tiveram comigo, em que muitas vezes não pude estar presente ou dando-lhes a atenção devida. Amo muito vocês!

Aos meus pais, José e Helena, que sempre acreditaram em mim e sempre estiveram ao meu lado, fornecendo suporte em todas as etapas da minha vida. Sou grato por todos os sacrifícios e desafios que vocês enfrentaram para que eu pudesse chegar até aqui.

Aos meus orientadores, professora Leticia Mara Peres e professor Marcos Didonet Del Fabro, muito obrigado pelo voto de confiança e paciência durante esses quatro anos. Aprendi muito durante nossas reuniões e escrita de artigos, sou extremamente grato por ter vivenciado essa experiência junto de vocês. Espero encontrá-los em breve.

Agradeço aos membros da banca examinadora, por compartilhar de seus conhecimentos, apontar pontos fortes e fracos do meu trabalho, e por apontar direcionamentos para trabalhos futuros. Fico feliz por ter a oportunidade de ser avaliado por um grupo de pesquisadores altamente capacitado.

Ao demais professores que contribuíram para minha formação, expresso minha gratidão por terem compartilhado de seus conhecimentos e experiências, que foram de grande valia para trilhar meu caminho. Como já dizia Augusto Cury, "professores brilhantes ensinam para uma profissão e professores fascinantes ensinam para a vida".

Agradeço aos meus amigos e colegas de trabalho da UTFPR pelo companheirismo e pelas várias horas de conversa durante as viagens e disciplinas do doutorado. Compartilhamos de boas histórias que levarei para sempre em minha memória. Foi muito bom tê-los ao lado durante essa etapa. Para finalizar, não posso deixar de agradecer o apoio da UTFPR - Dois Vizinhos, sem ele a condução do doutorado não seria possível.

RESUMO

Com o surgimento de novas aplicações surgiram também novos requisitos sobre os sistemas de armazenamento. Cenários envolvendo dados estruturados, semiestruturados e não-estruturados são cada vez mais comuns. Os bancos de dados relacionais (RDB, do inglês *Relational Database*), amplamente usados para armazenar dados de diversas aplicações, já não atendem de forma adequada todas as questões impostas pelos diferentes cenários. Como alternativa surgiram os bancos de dados NoSQL (do inglês, *Not only SQL*), flexíveis em relação ao modelo de dados e projetados para fornecer alta escalabilidade e disponibilidade. Bancos de dados relacionais e bancos de dados NoSQL coexistirão por longo período de tempo e, como consequência, novas abordagens para converter o modelo relacional para modelos de dados NoSQL foram propostas. No entanto, a maioria dessas abordagens se destina a conversão de dados relacionais para um modelo de dados NoSQL específico e fornecem pouco suporte para customizações do processo de conversão, como seleção de campos, tabelas, instâncias e outros aspectos relativos à customização do esquema de dados produzido. Além disso, há diversas formas de estruturar os dados (ou definir esquemas de dados) ao converter RDB para NoSQL. A escolha do esquema de dados adequado não é trivial e envolve vários aspectos, como o padrão de acesso aos dados, o nível de redundância de dados desejado, o tamanho do banco de dados NoSQL resultante, o esforço de manutenção da aplicação, dentre outros. Nesta tese é definida uma abordagem para converter e migrar dados relacionais para bases NoSQL orientadas a documentos e família de colunas, composta por uma etapa de avaliação de esquemas NoSQL candidatos. A abordagem usa grafos acíclicos direcionados (DAG, do inglês *Directed Acyclic Graph*) para especificar a estrutura das entidades que serão migradas para o modelo de dados NoSQL e, também, para representar o padrão de acesso da aplicação (consultas). Para avaliar a abordagem foram realizados experimentos envolvendo cenários de conversão de RDB para NoSQL compostos por diferentes esquemas NoSQL candidatos. Os resultados dos experimentos mostraram que a abordagem é eficaz para identificar cenários em que há maior esforço de implementação das consultas, auxiliando o usuário no processo de seleção de esquemas NoSQL, antes de migrar de dados.

Palavras-chave: Transformação de dados. Bancos de dados relacionais. Bancos de dados NoSQL. Conversão de bancos de dados. Métricas. Avaliação.

ABSTRACT

With the emergence of new applications, new requirements on storage systems have also emerged. Scenarios involving structured, semi-structured and unstructured data are increasingly common. Relational databases, widely used to store data from different applications, no longer adequately address all issues imposed by different scenarios. As an alternative, NoSQL databases have emerged, which are flexible in relation to the data model and designed to provide high scalability and availability. Relational databases and NoSQL databases will coexist for a long period of time and, as a consequence, new approaches to converting the relational model to NoSQL data models have been proposed. However, most of these approaches are aimed at converting relational data to a specific NoSQL data model and provide little support for customizing the conversion process, such as selection of fields, tables, instances, and other aspects related to the customization of the data schema produced. In addition, there are several ways to structure the data (or ways to define data schemas) when converting RDB to NoSQL. The choice of the appropriate data schema is not trivial and involves several aspects, such as the data access pattern, the desired level of data redundancy, the size of the resulting NoSQL database, the application maintenance effort, among others. This thesis defines an approach to convert and migrate relational data to document-oriented and column family NoSQL models, composed of an evaluation step of candidate NoSQL schemas. The approach uses directed acyclic graphs (DAG) to specify the structure of the entities that will be migrated to the NoSQL data model and also to represent the application's access pattern (queries). To evaluate candidate schemas, a set of metrics and scores was defined, which aims to measure the coverage of the NoSQL schema in relation to the set of queries. As NoSQL schema and query are defined through DAGs, it is possible to perform evaluations and comparisons objectively. To evaluate the approach, we performed experiments involving RDB to NoSQL conversion scenarios composed by different candidate NoSQL schemas. The results of the experiments showed that the approach is effective to identify scenarios in which there is a greater effort to implement the queries, assisting the user in the process of selecting NoSQL schemas, before executing the data migration.

Keywords: Data transformation. Relational databases. NoSQL databases. Database conversion. Metrics. Evaluation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
BASE	*Basically Available, Soft State, Eventually Consistent*
BSON	*Binary JSON*
CSV	Valores Separados por Vírgulas (*Comma-Separated-Values*)
DAG	Grafos Acíclicos Direcionados (*Directed Acyclic Graph*)
ETL	Extrair,Transformar, Carregar (*Extract, Transform, Load*)
FDU	Função Definida pelo Usuário
HDFS	*Hadoop Distributed File System*
IoT	Internet das Coisas (*Internet of Things*)
JSON	*JavaScript Object Notation*
LoC	Linhas de Código (*Lines of Code*)
MDA	Arquitetura Dirigida a Modelos (*Model-Driven Architecture*)
NoAM	*NoSQL Abstract Model*
NoSQL	Banco de Dados Não Relacional (*Not only SQL*)
OLAP	*On-line Analytical Processing*
QBM	Métricas-Baseadas em Consultas (*Query-Based Metrics*)
QODM	*Query-Oriented Data Modeling*
QScore	Escore da Consulta (*Query Score*)
RDB	Banco de Dados Relacional (*Relational Database*)
RDD	*Resilient Distributed Datasets*
SQL	Linguagem de Consulta Estruturada (*Structured Query Language*)
SScore	Escore do Esquema (*Schema Score*)
STD	Sistemas de Transformação de Dados
TLS	*Table-like-structure*
UML	Linguagem de Modelagem Unificada (*Unified Modeling Language*)
XML	*eXtensible Markup Language*
XQuery	Linguagem de Consulta para XML

LISTA DE SÍMBOLOS

C	Conjunto de coleções de esquema NoSQL documento
c	Uma coleção de documentos
E	Conjunto de arestas
E_c	Conjunto de arestas da coleção c
E_{dc}	Conjunto de arestas da coleção c, considerando direção das arestas
E_{dq}	Conjunto de arestas da consulta c, considerando direção das arestas
E_q	Conjunto de arestas da consulta q
Et	Entidade alvo de transformação de dados
G	Grafo
I	Instância de dados
I_s	Instância do esquema origem
I_t	Instância do esquema destino
M	Conjunto de mapeamentos
m_i	Mapeamento i
mt	Métrica
NC	Número de coleções requeridas para responder uma consulta
p	Caminho no grafo
P_c	Conjunto de caminhos da coleção c
p_{ind}	Caminho indireto no grafo
P_q	Conjunto de caminhos da consulta q
P_s	Conjunto de caminho do esquema s
Q	Conjunto de consultas
q	Consulta
qp	Caminho da consulta q
S	Esquema origem
S_{cf}	Esquema NoSQL orientado a família de colunas
S_{doc}	Esquema NoSQL orientado a documentos
T	Esquema destino
TB	Conjunto de tabelas de esquema NoSQL família de colunas
V	Conjunto de Vértices
V_c	Conjunto de vértices da coleção c
V_q	Conjunto de vértices da consulta q
V_s	Conjunto de vértices do esquema s
w_{ip}	Peso da métrica $IndirectPath$
w_p	Peso da métrica $Path$

w_{sp}	Peso da métrica $Subpath$

SUMÁRIO

1 INTRODUÇÃO

Por mais de quatro décadas o modelo relacional foi utilizado para armazenar os dados de diversos sistemas de informação. No entanto, com a popularização da computação em nuvem e o surgimento de novos tipos de aplicações, também surgiram novos requisitos sobre os sistemas de armazenamento de dados. Como exemplo, é possível citar as aplicações web e aplicações móveis, que produzem grande quantidade de dados durante a interação com o usuário, em formato estruturado, semiestruturado ou não-estruturado. Na troca de informações entre aplicações, muitas vezes, são usados formatos semiestruturados (ou hierárquicos), como XML (*eXtensible Markup Language*) ou JSON (*JavaScript Object Notation*), que encapsulam metadados e dados, facilitando a interoperabilidade entre as aplicações. Além disso, dependendo do domínio do problema, formatos semiestruturados correspondem diretamente à natureza dos dados manipulados, facilitando o desenvolvimento de soluções. Bancos de dados relacionais ou RDB (do inglês *Relational Database*) foram projetados para manipular dados com tipos de formatos restritos, não sendo capazes de gerenciar de maneira adequada todas as questões impostas pelas aplicações atuais (Stonebraker et al., 2007). O modelo relacional é baseado em relações, implementadas por meio de tabelas compostas por colunas e linhas em que sua estrutura precisa ser definida antes de iniciar o armazenamento de dados. Devido a essas características, os RDBs não são suficientemente flexíveis para armazenar dados de diversos formatos, o que torna a interoperabilidade e adaptabilidade difíceis. Além disso, exibem dificuldades em escalar horizontalmente, apresentando limitações quando o número de acessos concorrentes aumenta (Serrano et al., 2015; Stanescu et al., 2016).

Como alternativa surgiram os bancos de dados NoSQL (do inglês, *Not only SQL*), flexíveis em relação ao modelo de dados e projetados para fornecer alta escalabilidade horizontal e disponibilidade (Sadalage e Fowler, 2012). Diferem de bancos relacionais em termos de arquitetura, modelo de dados e linguagem de consulta. Os bancos de dados NoSQL podem ser classificados de acordo com o modelo de dados: chave-valor, documento, família de colunas e grafo. Cada um deles usa um modelo de dados distinto, que difere do modelo relacional e não tem suporte ao padrão SQL (do inglês, *Structured Query Language*), o que dificulta o processo de migração e adaptação de aplicações legadas (dos Santos Ferreira et al., 2013). Como estes tipos de bancos de dados coexistirão por longo período de tempo, é importante investigar abordagens de conversão e migração de dados entre eles.

1.1 JUSTIFICATIVA

Há diferentes soluções para converter o modelo relacional e dados para NoSQL. A maioria das abordagens segue uma metodologia baseada em duas etapas (Jia et al., 2016). Na primeira etapa, o modelo relacional é convertido em modelo NoSQL. Na segunda os dados são migrados de um banco para outro. Na etapa de conversão são definidos mapeamentos entre os conceitos de ambos os bancos de dados e regras de transformação para converter entidades do RDB para entidades NoSQL. Na etapa de migração é necessário estabelecer conexão com os bancos de dados, ler dados do banco origem, transformar e escrever dados no banco destino.

De forma geral, essas soluções desnormalizam os dados do RDB analisando as dependências entre tabelas e/ou padrão de acesso aos dados. Algumas delas executam algoritmos de conversão automáticos e não suportam customizações do processo de conversão, como seleção de tabelas, campos ou instâncias (Santos e Costa, 2016; Lee e Zheng, 2015; Vajk et al., 2013;

Zhao et al., 2014; Serrano et al., 2015; Stanescu et al., 2016; Zhao et al., 2014; Freitas et al., 2016). Outras soluções permitem ao usuário customizar a conversão das entidades, mas são soluções específicas que focam em apenas um modelo NoSQL (Karnitis e Arnicans, 2015; Jia et al., 2016). No entanto, nenhuma dessas soluções preocupa-se em avaliar se a estruturação dos dados gerada pela conversão é adequada aos requisitos da aplicação. Muitas vezes, o usuário especialista é a única garantia que o resultado produzido é adequado.

Devido à flexibilidade que os bancos de dados NoSQL fornecem em termos de estruturação dos dados, a escolha do formato adequado (ou esquema) não é uma tarefa trivial. A escolha depende de vários aspectos, como o padrão de acesso da aplicação, o nível de redundância de dados, o tamanho da base de dados, o esforço de manutenção da aplicação, entre outros. A Figura 1.1 ilustra essa situação, apresentando oito diferentes formas de estruturar as entidades *Orders*, *Orderlines* e *Products*, considerando bases NoSQL orientadas a documentos.

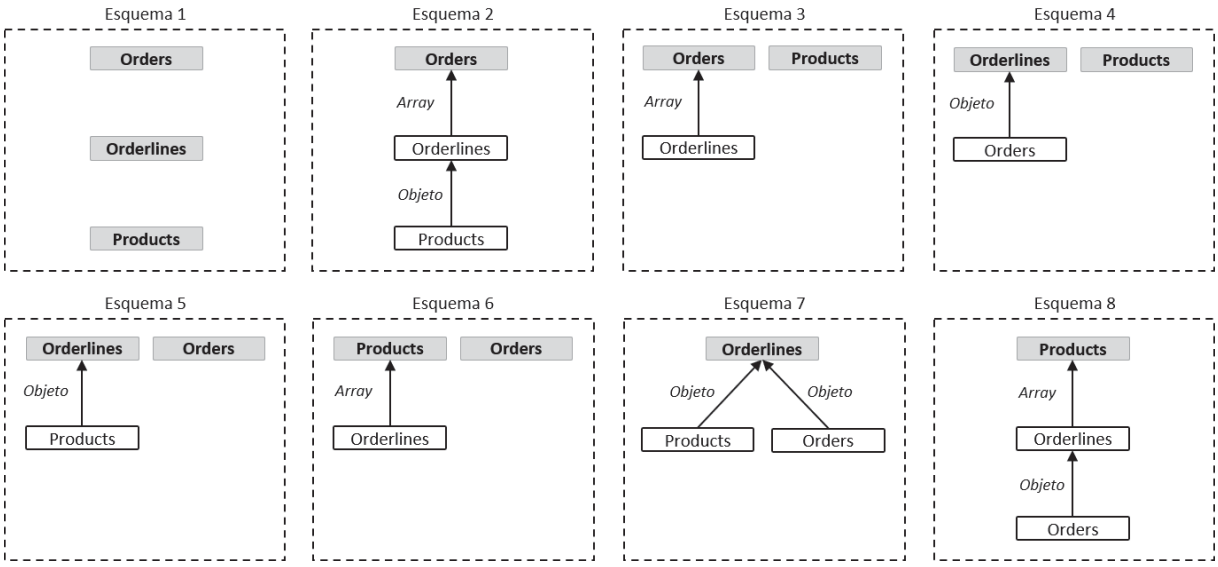


Figura 1.1: Conjunto de possíveis formas de estruturação de documentos a partir de três entidades.

Cada esquema é representado por meio de grafos acíclicos direcionados, em que os vértices cinza representam coleções de documentos e os vértices brancos os documentos aninhados. As arestas representam os relacionamentos entre as entidades e a direção indica a ordem de aninhamento. O relacionamento entre as entidades pode ser representado como referências, documentos embutidos ou *arrays* de documentos embutidos. Como pode ser visto na Figura 1.1, apenas no *Esquema 1* as entidades não estão aninhadas. Nos demais esquemas os relacionamentos são representados por documentos embutidos e *arrays* de documentos embutidos (ver anotação da aresta). Cada forma de estruturação de documentos tem vantagens e desvantagens, cabendo ao usuário decidir qual esquema melhor atende aos requisitos da aplicação.

Uma alternativa são as abordagens que usam o padrão de acesso da aplicação (consultas) para gerar um conjunto de esquemas NoSQL candidatos. NoSE (Mior et al., 2016) e QODM (Xiang Li et al., 2014) são exemplos de abordagens que seguem essa estratégia para recomendar esquemas. NoSE é voltado para bases NoSQL orientadas a famílias de colunas e QODM para bases orientadas a documentos. No entanto, ambas estratégias não fornecem meios de avaliar e comparar os possíveis esquemas gerados em relação aos demais requisitos da aplicação. Além disso, as abordagens não oferecem suporte para realizar a migração dos dados do RDB para NoSQL.

Outra alternativa é o uso de métricas para avaliar e comparar os esquemas candidatos antes de realizar a conversão e migração dos dados. O trabalho (Gómez et al., 2018) apresentou

onze métricas estruturais para avaliar esquemas NoSQL orientados a documentos, baseadas em métricas para avaliar esquemas XML apresentadas em (Pusnik et al., 2014) e (Klettke et al., 2002). Apesar de não ter um esquema formal, um documento tem uma estrutura usada pelas consultas para recuperar os dados. Essa estrutura é considerada como uma abstração para representar o esquema de bancos de dados NoSQL orientados a documentos. As métricas são computadas sobre a estrutura dos documentos, fornecendo informações para auxiliar o usuário especialista na seleção do esquema, conforme requisitos pré-definidos. No entanto, a abordagem não apresenta métricas específicas para avaliar se o padrão de acesso das consultas da aplicação é coberto pela estrutura do documento, que é um aspecto chave no processo de seleção do esquema de dados.

Esta tese aborda essa problemática por meio da definição de uma abordagem de conversão de RDB para NoSQL, baseada em etapas para definição de esquemas candidatos, para a avaliação, comparação e seleção de esquemas, e para migração dos dados. O foco da abordagem são as aplicações baseadas em operações de leitura, em que os dados devem estar estruturados na base NoSQL para atender certo padrão de acesso de leitura, como em aplicações OLAP (do inglês *On-line Analytical Processing*) e similares, em que os dados são escritos uma vez e lidos várias vezes.

1.2 OBJETIVOS

O objetivo desta tese é desenvolver uma abordagem de conversão de bases relacionais para bases NoSQL orientadas a documentos e famílias de colunas, com suporte à avaliação e comparação de esquemas NoSQL candidatos em relação ao padrão de acesso da aplicação.

Os objetivos específicos desta tese são os seguintes:

- Definir um processo para conversão e migração de bases relacionais para bases NoSQL, apoiada em uma abstração para representar o esquema de dados NoSQL destino (estruturação dos dados) e o padrão de acesso da aplicação (conjunto de consultas).
- Definir um conjunto de métricas para medir a cobertura que um determinado esquema NoSQL fornece para o padrão de acesso da aplicação.
- Definir um procedimento de uso das métricas para auxiliar no processo de avaliação, comparação e seleção de esquemas NoSQL candidatos, antes de realizar a migração de dados a partir da base relacional.
- Avaliar a abordagem de conversão definida e as ferramentas de suporte desenvolvidas por meio de experimentos.

1.3 CONTRIBUIÇÕES

A principal contribuição desta tese é uma abordagem para converter e migrar dados relacionais para bancos de dados NoSQL orientados a documentos e famílias de colunas. A abordagem é composta por quatro etapas e em cada uma delas são apresentadas contribuições específicas.

- Na primeira etapa são definidos os esquemas NoSQL candidatos para migrar RDB para NoSQL. A abordagem é baseada em grafos acíclicos direcionados (DAG, do inglês *Directed Acyclic Graph*) para especificar o processo de conversão de entidades RDB

para entidades NoSQL. Nesta etapa também são definidas as consultas que representam o padrão de acesso da aplicação. Esquemas e consultas são definidos como DAGs[1,2].

- A segunda etapa consiste na avaliação dos esquemas NoSQL candidatos em relação ao padrão de acesso da aplicação. Um conjunto de métricas é definido para avaliar a cobertura fornecida pelo esquema NoSQL em relação ao conjunto de consultas[1,2].

- A terceira etapa consiste na seleção do esquema NoSQL adequado aos requisitos da aplicação, em que o usuário especialista usa a abordagem para tomar a decisão. As três etapas iniciais da abordagem são realizadas por meio do *QBMetrics*, uma ferramenta desenvolvida para dar suporte à definição de esquemas e consultas por meio de DAGs, e computar as métricas[1,2].

- Na última etapa é realizada a migração de dados. O esquema NoSQL selecionado (conjunto de DAGs) é usado como especificação para o processo de migração de dados, sendo enviado ao Metamorfose, um *framework* de transformação de dados definido nesta tese. O Metamorfose lê os dados do RDB, transforma e persiste em formato NoSQL[3,4].

1.4 ORGANIZAÇÃO DO DOCUMENTO

Esta tese de doutorado está organizada da seguinte forma:

Capítulo 2 - Fundamentos: esse capítulo apresenta os conceitos relacionados a este trabalho, incluindo a definição de transformação de dados, correspondência e mapeamento entre esquemas de dados, conceitos sobre bancos relacionais e não-relacionais. Além disso, o modelo de programação *MapReduce* é apresentado em conjunto com o *framework* de processamento de dados *Apache Spark*. Por fim, são apresentados conceitos de métricas no contexto de avaliação de esquemas de dados.

Capítulo 3 - Estado da Arte: esse capítulo apresenta trabalhos relacionados à abordagem proposta neste trabalho. Foram consideradas abordagens para converter bancos de dados relacionais para bancos NoSQL orientados a famílias de colunas e orientados a documentos. Também foram apresentadas outras abordagens de modelagem de dados para bancos de dados NoSQL e uso de métricas para avaliar os modelos produzidos. Por fim, é realizada uma discussão sobre os principais trabalhos relacionados e suas limitações.

Capítulo 4 - Abordagem de Conversão de RDB para NoSQL e Migração de Dados: apresenta uma visão geral das etapas e da arquitetura da abordagem de conversão definida nesta tese. Nos próximos capítulos esses elementos serão detalhados.

Capítulo 5 - *Framework* Metamorfose para Conversão e Migração de Dados: apresenta uma visão geral do Metamorfose, um *framework* para converter e migrar bases relacionais

[1]Kuszera, E. M., Peres, L. M. e Didonet Del Fabro, M. (2020). Query-Based Metrics for Evaluating and Comparing Document Schemas. Em Dustdar, S., Yu, E., Salinesi, C., Rieu, D. e Pant, V., editores, Advanced Information Systems Engineering, páginas 530–545, Cham. Springer International Publishing.

[2]Kuszera, E. M., Peres, L. M. e Didonet Del Fabro, M. (2020). QBMetrics: A Tool for Evaluating and Comparing Document Schemas. Em Herbaut, N. e La Rosa, M., editores, Advanced Information Systems Engineering - CAiSE Forum, páginas 77–85, Cham. Springer International Publishing.

[3]Kuszera, E. M., Peres, L. M. e Fabro, M. D. D. (2018). Metamorfose: a Data Transformation Framework Based on Apache Spark. Em 33rd Annual Brazilian Symposium on Databases: Proceeedings Companion, SBBD 2018 Companion, Rio de Janeiro, RJ, Brazil, August 25-26, 2018., páginas 11–16.

[4]Kuszera, E. M., Peres, L. M. e Fabro, M. D. D. (2019). Toward RDB to NoSQL: Transforming Data with Metamorfose Framework. Em Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19, página 456–463, New York, NY, USA. Association for Computing Machinery.

para bases de dados NoSQL, incluindo a arquitetura e o processo de conversão dos DAGs em comandos para transformar os dados do RDB para formato NoSQL.

Capítulo 6 - Experimentos: Uso do Metamorfose para Conversão e Migração de Dados: apresenta os experimentos realizados para validar o *framework* no contexto de transformação de dados relacionais e no contexto de transformação de dados relacionais para base de dados NoSQL.

Capítulo 7 - Métricas Baseadas em Consultas: esse capítulo apresenta o conjunto de métricas baseadas em consultas (QBM, do inglês *Query-Based Metrics*). As métricas têm por objetivo medir a cobertura que um esquema NoSQL documento tem em relação a um conjunto de consultas. Nesse capítulo também são apresentadas uma ferramenta para dar suporte no uso das métricas e diretrizes para implementar consultas na linguagem do banco NoSQL.

Capítulo 8 - Experimentos: Usando Métricas Baseadas em Consultas: esse capítulo apresenta os experimentos realizados para validar o conjunto de métricas baseadas em consulta (QBM). Dois experimentos foram realizados. O primeiro experimento aplica as métricas em um cenário de conversão de RDB para NoSQL orientado a documentos, em que vários esquemas NoSQL candidatos são avaliados em relação a um conjunto de consultas previamente definido. O segundo experimento tem por objetivo avaliar o impacto da localização do filtro da consulta em relação ao esforço de implementação e tempo de execução em diferentes esquemas.

Capítulo 9 - Conclusões: esse capítulo apresenta as considerações finais sobre esta tese de doutorado, incluindo informações sobre trabalhos futuros. Na sequência, são apresentadas as referências bibliográficas usadas neste documento.

2 FUNDAMENTOS

Este capítulo apresenta a fundamentação teórica necessária para auxiliar no entendimento desta tese de doutorado. São apresentados conceitos sobre bancos relacionais e não-relacionais, transformação de dados, correspondência e mapeamento de esquema, que são o foco principal desta tese. O modelo de programação *MapReduce* e o *framework* de processamento de dados distribuído *Apache Spark* são apresentados. Ambos são usados na abordagem de transformação de dados apresentada nesta tese. Por último, são introduzidos conceitos de métricas no contexto de avaliação de esquemas.

2.1 BANCOS DE DADOS RELACIONAIS E NÃO-RELACIONAIS

A maioria das aplicações utilizam o modelo relacional para armazenar dados. Esse modelo representa os dados como tuplas agrupadas em relações, com uma forte fundamentação matemática (Codd, 1970). Fornece independência de dados, desacoplando as representações lógica e física dos dados. Os dados são organizados em tabelas compostas por colunas e linhas. O controle de transações (conjunto de uma ou mais operações relacionadas) é baseado nas propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade), que garantem que as transações são atômicas, consistentes, independentes e duráveis. Bancos de dados baseados em ACID visam consistência e integridade dos dados acima de outras considerações, usando bloqueios (*locks*) para garantir que as informações retornadas são confiáveis e acuradas. O usuário interage com o banco de dados através da *Structured Query Language* (SQL), uma linguagem para definir, manipular e consultar dados de forma declarativa (Astrahan e Chamberlin, 1975). Devido à facilidade de uso da linguagem SQL e capacidades do modelo relacional, diferentes sistemas foram implementados ao longo das últimas décadas, e ainda são implementados usando essas tecnologias.

No entanto, as aplicações atuais criaram demandas novas sobre os sistemas de armazenamento de dados. Segundo (Davoudian et al., 2018), aplicações web, redes sociais, dispositivos móveis e internet das coisas (IoT, do inglês *Internet of Things*) resultaram em uma explosão de dados estruturados, semiestruturados e não-estruturados. Essas aplicações têm requisitos como escalabilidade horizontal, alta disponibilidade, tolerância a falhas, além de lidar com alta heterogeneidade dos dados. Atender tais requisitos por meio dos bancos relacionais é desafiador. Primeiro, os bancos relacionais necessitam de um esquema de dados pré-definido, o que pode dificultar o desenvolvimento de aplicações que evoluem rapidamente. Segundo, escalar verticalmente bancos relacionais necessita mover os dados para nova infraestrutura de hardware, geralmente de alto custo. Por último, escalar horizontalmente (distribuir em vários computadores) bancos relacionais é complexo, devido as propriedades ACID.

Em função disso surgiram os bancos de dados NoSQL (do inglês, *Not only SQL*) que têm por objetivo alta disponibilidade e escalabilidade, além de lidar com dados heterogêneos (Sadalage e Fowler, 2012). Esses bancos de dados não seguem o modelo relacional, não têm esquema pré-definido (o esquema está encapsulado na aplicação), não usam a linguagem SQL e geralmente são projetados para operar em agregados de computadores (*clusters*). Diferente do modelo relacional, os bancos de dados NoSQL são baseados nas propriedades BASE (*Basically Available*, *Soft State*, *Eventually Consistent*) (Pritchett, 2008). Isso significa que o sistema está disponível a maior parte do tempo (basicamente disponível), o seu estado pode estar temporariamente inconsistente (estado-leve) e eventualmente, após a execução de toda a lógica dos serviços, o

estado do sistema estará consistente (eventualmente consistente). Essas propriedades permitem implementar bancos de dados escaláveis, em detrimento da garantia imediata da consistência dos dados.

Os bancos de dados NoSQL não substituem os bancos relacionais, mas oferecem uma opção para cenários em que os bancos relacionais apresentam deficiências. Geralmente, os bancos de dados NoSQL são divididos em quatro categorias (Sadalage e Fowler, 2012): chave-valor, grafo, família de colunas e documento.

Os bancos de dados orientados a chave-valor estruturam seus dados na forma de pares chave-valor. São implementados por meio de uma tabela *hash*, em que a chave representa o índice e ponteiro para o dado armazenado. O dado armazenado pode ser simples (dados numéricos ou cadeias de caracteres) ou estruturado (objetos JSON, por exemplo). Como não há suporte a esquema de dados, o usuário pode nomear as chaves de forma a introduzir manualmente metadados. Além disso, não são suportadas consultas sobre os dados, apenas sobre as chaves. Também não há suporte para relacionamentos e restrições de integridade referencial. Redis[1] e Riak[2] são dois exemplos de bancos de dados dessa categoria.

Bancos NoSQL orientados a grafos armazenam as entidades como uma sequência de nós e relacionamentos. São baseados em três campos de dados: nós, relacionamentos e propriedades. As entidades são representadas como nós e têm propriedades. Os relacionamentos são representados como arestas e têm propriedades. A direção das arestas tem significado e os nós são organizados através dos relacionamentos. O grafo resultante pode ser consultado de várias formas, permitindo descobrir padrões sobre os dados relacionados. Esse tipo de banco de dados é ideal quando há muitos itens que estão relacionados uns com os outros de uma maneira complexa. Neo4J[3] é um exemplo de banco NoSQL dessa categoria.

Nesta tese os bancos de dados NoSQL orientados a documentos e famílias de colunas são usados como destino das implementações de transformação de dados relacionais para NoSQL do Capítulo 5. As próximas seções introduzem essas duas categorias de bancos de dados NoSQL.

2.1.1 NoSQL Orientado a Família de Colunas

Bancos de dados NoSQL orientados a família de colunas organizam os dados em tabelas compostas por famílias de colunas. No entanto, a estrutura dos dados pode ser melhor vista como um agregado de dados organizados em dois níveis (Sadalage e Fowler, 2012). O primeiro nível consiste no identificador do registro, geralmente chamado de *rowkey*. O segundo nível consiste nos identificadores das colunas. Por meio do identificador do registro e identificador da coluna é possível acessar o valor armazenado no banco de dados. Esses bancos de dados organizam as colunas em famílias de colunas, de forma que cada família de colunas armazena seus dados fisicamente próximos, assumindo que são acessados frequentemente juntos. A Figura 2.1 exibe a tabela *Customers* representada por meio do modelo orientado a família de colunas. Neste exemplo, há duas famílias de colunas chamadas *Profile* e *Orders*, que agrupam dados relacionados. O campo *rowkey* é o identificador do *Customer*.

Um registro pode estar associado a várias famílias de colunas e estar associado a um número diferente de colunas, não sendo necessário reservar espaço de armazenamento para valores nulos. Cassandra[4] e HBase[5] são exemplos de bancos NoSQL dessa categoria.

[1] https://redis.io/
[2] https://riak.com/
[3] https://neo4j.com/
[4] https://cassandra.apache.org/
[5] http://hbase.apache.org/

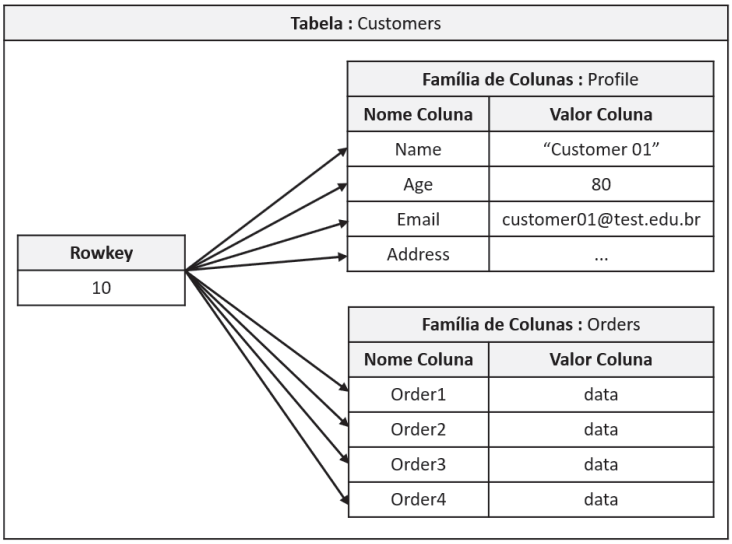


Figura 2.1: Relacionamento entre *Customer* e *Orders* usando o modelo orientado a famílias de colunas.

2.1.2 NoSQL Orientado a Documentos

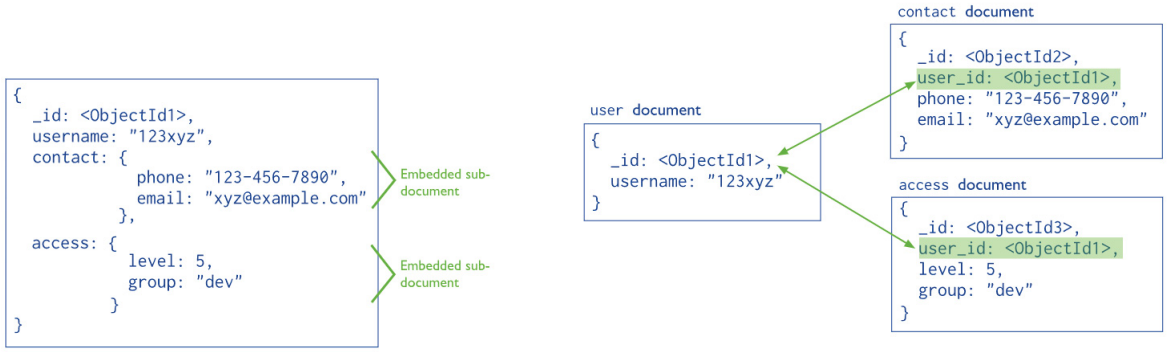
Nesta categoria, os dados são armazenados como documentos. Um documento é uma estrutura hierárquica e auto-descritiva, composto por campos no formato chave-valor. A chave é um identificador do campo e o valor pode armazenar mapas, coleções e valores escalares. Os formatos dos documentos armazenados são geralmente XML, JSON e BSON (*Binary JSON*). O código abaixo apresenta a estrutura de um documento JSON com informações sobre uma entidade *cliente*. O documento é composto pelos campos *_id*, *name*, *age*, *email*, *address* e *phone*. O campo *_id* é o identificador do documento, semelhante ao conceito de chave primária em bancos relacionais. O campo *address* armazena um sub-documento ou documento embutido com as informações do endereço do cliente. No campo *phone* é armazenado um *array* de documentos embutidos, cada um representando um telefone de contato do cliente.

```
{
  "_id": 10,
  "name": "Customer 01",
  "age": 80,
  "email": "customer01@test.edu.br",
  "address": {
    "street": "Castro Alves, 100",
    "zip": "85660-000",
    "city": "Dois Vizinhos",
    "state": "Paraná"
  },
  "phone": [
    {"type": "celphone", "number": "123456789"},
    {"type": "home_phone", "number": "987654321"}
  ]
}
```

Os bancos de dados NoSQL orientados a documentos geralmente organizam os documentos por meio de coleções, agrupando documentos semelhantes, de forma que um banco de dados é composto por várias coleções. No entanto, cada documento tem seu próprio conjunto de campos, de forma que uma coleção pode armazenar documentos que diferem em termos estrutura. Diferentemente dos bancos NoSQL chave-valor, é possível executar consultas mais complexas, envolvendo diferentes coleções de documentos (referências) e filtros sobre os valores

dos campos do documento. Apesar de permitir referências, não há suporte às restrições de integridade referencial.

O relacionamento entre entidades é representado por meio de documentos embutidos, *arrays* de documentos embutidos ou referências (semelhante ao modelo relacional). A escolha de qual representação usar depende dos requisitos da aplicação. A Figura 2.2 exibe exemplos de como representar os relacionamentos entre as entidades *User*, *Contact* e *Access* por meio de documentos embutidos e referências.



(a) Documentos embutidos (b) Referências

Figura 2.2: Formas de representar relacionamentos em bancos de dados orientados a documentos. Fonte: MongoDB.

MongoDB[6] e CouchBase[7] são exemplos de bancos NoSQL orientados a documentos.

2.2 TRANSFORMAÇÃO DE DADOS E CONVERSÃO DE BASES DE DADOS

Os dados em sistemas computacionais podem ser estruturados, semiestruturados e não-estruturados (Elmasri e Navathe, 2011). Dados não-estruturados não têm uma estrutura definida *a priori*, como por exemplo arquivos de textos, áudio e vídeo. Esses dados precisam de um pré-processamento para que informações possam ser extraídas. Dados estruturados têm um esquema previamente definido, em que todas as ocorrências do dado (ou instâncias) seguem esse esquema. Os bancos de dados relacionais armazenam dados estruturados, de forma que todo registro de uma tabela tem o mesmo número de campos. Os dados semiestruturados apresentam estrutura heterogênea, em que uma ocorrência do dado pode ser incompleta ou irregular em relação a outra ocorrência do dado, com atributos adicionais ou ausência de atributos. Os dados semiestruturados são também chamados de auto descritivos, pois encapsulam informações do esquema (nome dos atributos, relacionamentos e tipos de entidade) junto com os valores dos dados. Arquivos XML e JSON são exemplos de dados semiestruturados.

Em função da diversidade de formatos de dados usados por aplicações existentes, é comum a necessidade de transformar dados de um formato para outro.

2.2.1 Transformação de Dados

O processo de alterar o formato original dos dados para outro formato é conhecido como transformação de dados. Esse processo é utilizado em vários contextos, como em integração de dados, gerenciamento de dados, migração de dados e construção de armazéns de dados (do inglês, *data warehouse*). Dependendo do cenário e da natureza dos dados, uma transformação

[6] https://www.mongodb.com/
[7] http://couchdb.apache.org

pode ser composta por várias tarefas, simples ou complexas, envolvendo passos automáticos e passos manuais.

Sistemas de transformação de dados (STD) são importantes nesses cenários. A função desses sistemas é executar transformações nas quais instâncias de um esquema origem são traduzidas para instâncias de um esquema destino. STDs são compostos por um conjunto de tarefas de tradução que podem ser definidas como uma quadrupla $\{S, T, I_s, I_e\}$, em que S é o esquema de origem, T é o esquema de destino, I_s é uma instância válida de S e I_e uma instância válida de T gerada aplicando as transformações desejadas sobre I_s (Mecca et al., 2012). Geralmente para realizar transformações, um conjunto de mapeamentos M é empregado para denotar o relacionamento entre a saída esperada e a entrada fornecida. A Figura 2.3 exibe uma representação de STDs.

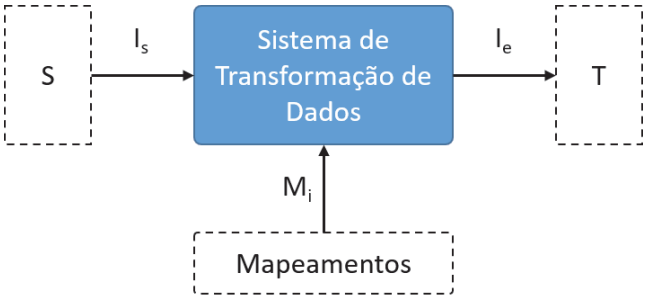


Figura 2.3: Representação de sistemas de transformação dados.

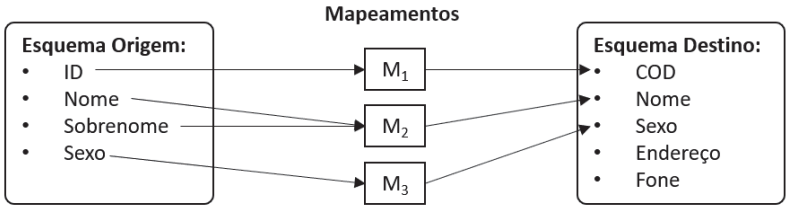
Para realizar as transformações de dados é necessário definir os mapeamentos entre os campos do esquema de origem e esquema de destino. Esses mapeamentos definem como serão produzidas as saídas da transformação. Em relação ao tipo de mapeamento há duas possibilidades (Scavuzzo et al., 2014): mapeamento direto e mapeamento intermediário. No mapeamento direto as estruturas de dados de origem são convertidas diretamente para as estruturas de dados de destino. No mapeamento intermediário as estruturas de dados de origem são convertidas para um formato intermediário e depois para o formato destino. Geralmente o mapeamento intermediário apresenta desempenho inferior, pois necessita de duas transformações para gerar o modelo de dados destino. No entanto, fornece maior flexibilidade para adicionar novos modelos de dados a solução de transformação de dados, pois são necessários dois mapeamentos: *i)* do novo modelo para o modelo intermediário e, *ii)* do modelo intermediário para o novo modelo.

2.2.2 Correspondência e Mapeamento de Esquema

Mapeamento de esquema (do inglês, *schema mapping*) é um conjunto de expressões que descreve o relacionamento entre dois ou mais esquemas (Doan et al., 2012). O mapeamento de esquema (ou simplesmente mapeamento) é usado em vários contextos, como integração de dados, *data exchange* e *data warehouse*. Na integração de dados, o mapeamento é usado para descrever o relacionamento entre um esquema intermediário e vários esquemas de origem. Quando uma consulta é submetida ao esquema intermediário, os mapeamentos são usados para reformular essa consulta, de forma que seja possível submetê-la diretamente sobre as fontes de dados. No contexto de *data exchange* e *data warehousing*, os mapeamentos expressam relacionamentos entre um banco de dados de origem e um banco de dados de destino. Os mapeamentos são usados para mapear os dados de um banco para outro, permitindo transformar os dados do formato de origem para o formato de destino.

A Figura 2.4 apresenta um exemplo de transformação de dados sobre uma tabela com dados de pessoas. O campo ID é mapeado para o campo COD no esquema destino. Os campos

NOME e SOBRENOME são mapeados para o campo NOME no esquema destino. Os dados do campo SEXO são transformados de 'F' e 'M' para os valores numéricos 1 e 2, respectivamente. Os campos ENDEREÇO e FONE existem apenas no esquema destino, em que ENDEREÇO tem como valor padrão uma *string vazia* e FONE o valor *null*. A semântica das transformações e especificação dos mapeamentos podem ser expressas como consultas (SQL ou XQuery), como dependências lógicas ou como especificações procedurais. Nesse último caso, as especificações procedurais são geralmente usadas por ferramentas de ETL (do inglês *Extract,Transform,Load*) (Mecca et al., 2012).



Entrada:

ID	Nome	Sobrenome	Sexo
1	João	Silva	M
2	Maria	Silva	F
3	José	Silva	M

Transformação de dados

Saída:

COD	Nome	Sexo	Endereço	Fone
1	João Silva	1	""	*null*
2	Maria Silva	2	""	*null*
3	José Silva	1	""	*null*

Figura 2.4: Exemplo de mapeamento entre esquemas.

CLIO (Haas et al., 2005) foi o primeiro sistema que abordou o problema de gerar mapeamentos a partir de um conjunto de correspondências entre esquemas. Através de uma interface gráfica o usuário estabelece as correspondências entre esquemas. Essas correspondências são compiladas na forma de um grafo que captura a semântica das transformações. O grafo resultante pode ser serializado em diferentes linguagens de consultas, como SQL ou XQuery, dependendo das fontes de dados de origem e destino. O objetivo das consultas é retornar instâncias do esquema de destino com base nas instâncias do esquema origem.

2.2.3 Conversão de Bases de Dados

O processo de conversão pode ser realizado entre bases de dados que usam um modelo de dados em comum ou pode ser realizada entre bases de dados que usam modelos de dados distintos (como os modelos relacional e orientado a documentos). De forma geral, a conversão entre bases de dados consiste em três etapas: *(1)* definição de mapeamentos de conceitos de ambos os modelos; *(2)* definição de regras de transformação; e *(3)* execução do processo de conversão e migração dos dados. A Figura 2.5 ilustra as etapas de conversão entre bases de dados.

Quando os dois bancos de dados usam o mesmo modelo de dados (modelo relacional, por exemplo), a etapa de mapeamento de conceitos é direta, em que cada conceito do modelo de origem tem conceito correspondente no modelo de destino, por exemplo, tabelas são mapeadas para tabelas e colunas para colunas. No entanto, quando há necessidade de converter bases de dados que usam diferentes modelos de dados (modelo relacional para modelo NoSQL orientado a documentos, por exemplo) é necessário definir como cada conceito do banco de origem se relaciona (equivalência ou ausência, por exemplo) com os conceitos do banco de destino (etapa *1*). Na etapa *2* é necessário definir operações para transformar uma instância de dados do modelo

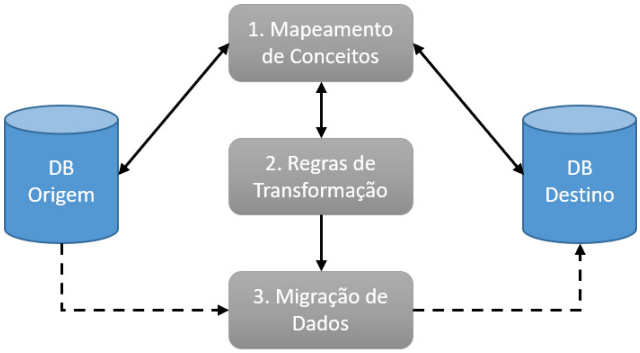


Figura 2.5: Processo de conversão entre bases de dados.

de origem para uma ou mais instâncias do modelo de destino, considerando o mapeamento de conceitos. Regras para definir identificadores e converter os relacionamentos entre as instâncias são definidas, considerando os metadados de cada modelo. Na etapa *3* a conversão é executada e os dados são migrados, na qual as instâncias são recuperadas da origem, são transformadas conforme as regras e inseridas na base de destino. STDs geralmente são usados nas etapas *2* e *3* para definir as transformações e executar a migração dos dados.

Esses conceitos foram empregados na implementação do Metamorfose, um *framework* de transformação de dados definido no Capítulo 5. Os conceitos serviram de base para definir as especificações das transformações de dados entre origem e destino.

2.3 MODELO DE PROGRAMAÇÃO *MAPREDUCE*

MapReduce é uma abstração inspirada nas primitivas *map* e *reduce* encontradas em linguagens de programação funcionais, como LISP. Essa abstração permite expressar tarefas para serem executadas de forma paralela e distribuída, porém escondendo do usuário detalhes como tolerância a falhas, distribuição de dados e balanceamento de carga (Dean e Ghemawat, 2008). Esse modelo de programação foi concebido pela Google [8] e usado para processar grandes quantidades de dados usando máquinas heterogêneas, considerando a ocorrência de falhas durante as computações.

Para desenvolver programas *MapReduce* o usuário deve expressar as computações através de duas funções: *map* e *reduce*. De forma geral, um programa *MapReduce* recebe um conjunto de pares de chave-valor e retorna outro conjunto de pares de chave-valor. Para cada par consumido pela função *map* é emitido um conjunto de pares de chave-valor intermediário. Os valores intermediários são agrupados de acordo com a chave intermediária e encaminhados para a função *reduce*. A função *reduce* recebe a chave e conjunto de valores associados com essa chave. O usuário deve implementar a lógica de redução dos dados, na qual a função *reduce* pode retornar zero ou mais pares de chave-valor. Através desse modelo de programação é possível expressar vários tipos de computações.

Para exemplificar o modelo de programação *MapReduce*, considere uma aplicação para contar a ocorrência de palavras em uma grande coleção de documentos. A Figura 2.6 exibe o fluxo de execução da aplicação. A função *map* recebe um conjunto de arquivos contendo palavras e gera um par de chave-valor para cada palavra. Nesse exemplo, a palavra representa a chave e o valor representa uma ocorrência dessa palavra (valor igual a 1). Na fase de *shuffle* os valores são agrupados de acordo com a chave (palavra) e armazenados como uma lista de valores. A função

[8] `https://www.google.com`

reduce processa essa lista e calcula o número de ocorrências de cada palavra. Depois que todas as funções *map* e *reduce* são executadas, o resultado é retornado para o programa do usuário.

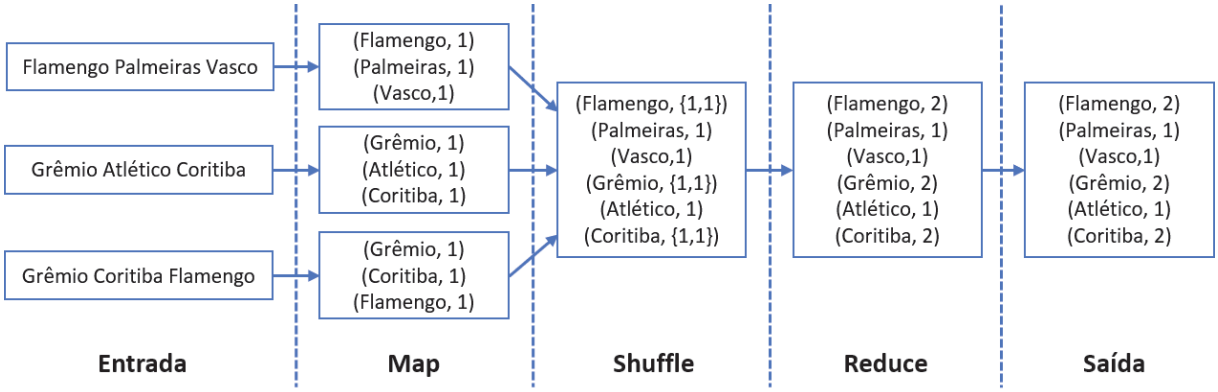


Figura 2.6: Fluxo de execução de uma aplicação *MapReduce* para contar palavras contidas em documentos.

Apache Hadoop[9] e *Apache Spark* (Zaharia et al., 2016) são duas implementações bem conhecidas do modelo de programação *MapReduce*. *Apache Hadoop* é baseado em uma arquitetura mestre-escravo, em que os nós mestres gerenciam o armazenamento de dados e execução das computações. Os dados são armazenados no *Hadoop Distributed File System* (HDFS). No *Apache Hadoop*, cada nó realiza computações e armazena dados produzidos, procurando levar a computação em direção aos dados, evitando transferência de dados pela rede. No entanto, o resultado de cada função *map* e *reduce* é armazenado no HDFS, limitando o desempenho das aplicações em função do custo de operações de entrada/saída. Devido a esse modelo de persistência surgiu o *Apache Spark* como alternativa ao *Apache Hadoop*.

2.3.1 *Apache Spark*

Apache Spark é um *framework* unificado para processamento de dados distribuídos (Zaharia et al., 2016). Ele fornece um modelo de programação similar ao *MapReduce*, mas estende esse modelo por meio de uma abstração de dados compartilhados chamada de RDD (do inglês *Resilient Distributed Datasets*). Através do RDD é possível capturar um amplo conjunto de cargas de trabalho sem precisar desenvolver *frameworks* específicos.

Um RDD é uma coleção de objetos tolerantes a falhas particionada sobre um *cluster* de computadores que pode ser manipulada em paralelo (Zaharia et al., 2012). RDDs podem ser criados a partir de dados armazenados ou a partir de outros RDDs. As operações sobre os RDDs são agrupadas em operações de transformação e operações de ação. As operações de transformação, como *map*, *filter* e *groupBy* definem como gerar um RDD de destino a partir de um RDD de origem, mas não disparam a execução de computações para processamento dos dados. A execução das transformações de dados é disparada pelas operações de ação, como *count*, *collect* e *persist*. O resultado de uma operação de ação pode ser retornado para o programa do usuário ou enviado para um sistema de armazenamento externo. Esse processo de avaliação de RDD é chamado de preguiçoso (do inglês *lazy*). Somente quando é chamada uma operação de ação é que as transformações são executadas. Esse mecanismo permite ao *Spark* otimizar o plano de execução das transformações de dados, rearranjando a ordem das transformações ou agrupando transformações quando possível.

Uma aplicação *Spark* consiste em um programa do usuário (também conhecido como *driver program*) e um conjunto de nós trabalhadores. O programa do usuário se conecta com um

[9] `http://hadoop.apache.org`

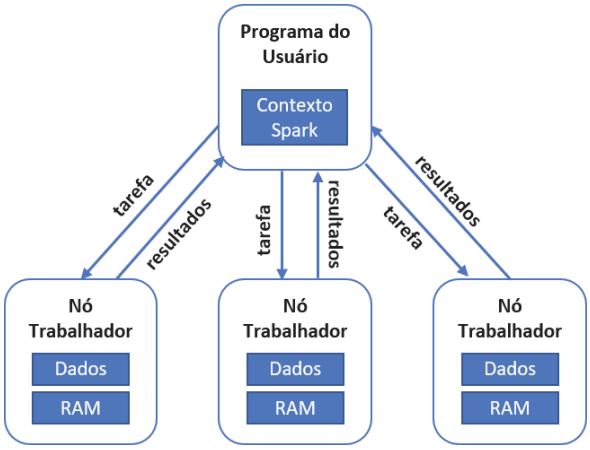


Figura 2.7: Componentes de uma aplicação *Apache Spark* sobre um *cluster* de computadores.

aglomerado de nós trabalhadores, conforme ilustrado na Figura 2.7. Através do programa do usuário é possível definir um ou mais RDDs e invocar operações de transformação e ação sobre eles. Os nós trabalhadores são processos que podem armazenar partições RDD na memória RAM e executar computações sobre os dados. Ao final, os resultados das computações são enviados dos nós trabalhadores para o programa do usuário. Essa arquitetura permite executar operações *MapReduce* de forma distribuída e paralela. No entanto, é possível executar todos os componentes em apenas uma máquina, permitindo realizar testes antes de submeter o programa para um *cluster* de computadores. De forma diferente ao *Apache Hadoop*, os dados são armazenados na memória principal do nó trabalhador, somente sendo persistidos em disco caso não caibam na memória. Essa característica permitiu ao *Apache Spark* obter desempenho superior ao *Apache Hadoop* (Zaharia et al., 2012).

O *Spark* fornece suporte para integrar com diversos sistemas de armazenamento. O usuário pode criar RDDs a partir de arquivos em disco, bancos relacionais e bancos NoSQL. A partir da abstração de dados RDD foram criadas diversas bibliotecas especializadas de processamento de dados. As principais bibliotecas são (Zaharia et al., 2016):

- SQL e *Dataframes*: através do *Spark SQL* é possível submeter consultas SQL sobre um RDD. Essa biblioteca tem outra abstração de dados chamada *DataFrame*. Um *Dataframe* representa dados tabulares e tem esquema de dados conhecido. Isso permite aplicar otimizações no acesso aos dados.

- *Streaming*: implementa processamento de fluxo (do inglês, *stream*) incremental, em que os dados de entrada são divididos em pequenos lotes e regularmente combinados como os dados do RDD existente para produzir novos resultados.

- MLlib: biblioteca que implementa mais de 50 algoritmos de aprendizado de máquina de modelo de treinamento distribuído.

- GraphX: fornece uma interface de computação para grafos.

Em função das características apresentadas acima, o *Apache Spark* é utilizado como *framework* de processamento de dados adjacente do Metamorfose. As principais características consideradas são a possibilidade de manipular vários formatos de dados e executar transformações de dados de forma distribuída e paralela através do modelo de programação *MapReduce*.

2.4 MÉTRICAS PARA AVALIAÇÃO DE ESQUEMAS DE BANCOS DE DADOS

Uma métrica é um valor numérico para um atributo de uma determinada entidade, em que esse valor é usado para avaliação ou comparação com valores ou padrões previamente estabelecidos. As métricas podem ser usadas para estimar, prever e avaliar diferentes características. No contexto da computação, uma entidade pode ser um *software*, um processo de produção do *software*, um arquivo XML, um esquema de dados, dentre outros.

Engenharia de *software* é uma das áreas da computação em que foram definidas diversas métricas. De acordo com (IEEE Std 610.12, 1990), uma métrica é uma medida quantitativa do grau com o qual um sistema, componente ou processo, tem certo atributo. Por meio da análise dos valores medidos é possível inferir aspectos de qualidade do *software* ou do processo de desenvolvimento, como manutenibilidade, confiabilidade, reusabilidade e usabilidade (Timóteo et al., 2008). As métricas são divididas em métricas dinâmicas, aferidas com o *software* em execução, e métricas estáticas, que são aferidas analisando artefatos, como documentos, modelos, diagramas, entre outros. Exemplos de métricas estáticas que podem ser usadas são (Sommerville, 2011):

- *Fan-in/Fan-out*: *fan-in* mede o número de funções ou métodos que chamam uma função ou método *X*, em que um *fan-in* alto para *X* significa que a função está fortemente acoplada ao sistema. *Fan-out* mede o número de funções que são chamadas por *X*, em que um número alto significa que *X* pode apresentar alta complexidade de controle lógico.

- Comprimento de código: mede o tamanho do código, geralmente em linhas de código (LoC, do inglês *Lines of Code*). À medida que o código aumenta, a complexidade e dificuldade em manter o código fonte também aumentam.

- Complexidade ciclomática: mede a complexidade de controle de um programa e está associada à compreensibilidade do programa.

- Comprimento de identificadores: mede o tamanho médio dos identificadores do programa, como nome de variáveis, métodos, funções, classes, dentre outros. Identificadores longos possivelmente são mais legíveis e aumentam a compreensibilidade do *software*.

- Profundidade de aninhamento condicional: mede a profundidade de aninhamento de declarações *if*, em que declarações com vários níveis de aninhamentos são difíceis de entender e sujeitas a erros.

- Índice *fog*: mede o tamanho médio das palavras e sentenças em documentos, na qual um valor alto de índice *fog* indica maior dificuldade na compreensão do documento.

Documentos XML são usados no contexto de desenvolvimento de *software*, seja para definir a estrutura de dados interna do *software* ou para estabelecer interoperabilidade entre diferentes sistemas. Desta forma, os documentos XML influenciam no desenvolvimento de *software* e é interessante poder medir a qualidade deles (Misra et al., 2017). Geralmente as métricas relacionadas a documentos e esquemas XML são adaptações das métricas definidas na engenharia de *software*. Há métricas relacionadas ao tamanho do esquema XML, que contam o número de elementos, atributos, entidades e notações. Há métricas para avaliar a complexidade estrutural do esquema XML, em que o esquema é representado como um grafo e a complexidade é calculada com base no número de vértices e arestas. Essas métricas são usadas para avaliar a complexidade dos esquemas (Klettke et al., 2002). Há também métricas para avaliar a qualidade

de esquemas XML, incluindo aspectos como estrutura, clareza, otimalidade, minimalismo, reuso e flexibilidade (Pusnik et al., 2014). Exemplos de métricas que podem ser usadas são (Klettke et al., 2002):

- Tamanho: mede o número de elementos e atributos do esquema XML.
- Complexidade estrutural: é baseada na complexidade ciclomática da engenharia de *software*, em que o esquema XML é representado como um grafo e a métrica é calculada como $num\_arestas - num\_vertices + 1$. Quanto maior o valor, maior é a complexidade estrutural do esquema.
- Profundidade: mede a profundidade do esquema XML (grafo), em que esquemas com profundidade maior são mais complexos.
- *Fan-in/Fan-out*: similar as métricas *fan-in/fan-out* da engenharia de *software*. *Fan-in* mede o número de vértices filho de um determinado vértice, em que um valor maior indica que o elemento é mais complexo e difícil de entender do que os demais. *Fan-out* mede o número de vértices pai que um determinado vértice tem, em que um valor maior indica que o elemento é reusado várias vezes, requerendo maior esforço ao alterar o esquema XML.

Na literatura, diferentes métricas foram propostas para avaliar a qualidade de esquemas de dados relacionais (Moody, 1998) e multidimensionais (Di Tria et al., 2017). Em (Moody, 1998) foram propostas métricas para avaliar o desempenho e qualidade de esquemas relacionais, incluindo métricas objetivas que contam número de elementos do esquema conceitual (entidades e relacionamentos, por exemplo), e métricas subjetivas que envolvem os interessados (do inglês, *stakeholders*) do projeto. Essas métricas são usadas para avaliar se o esquema é considerado completo, íntegro, simples, correto, flexível, de fácil implementação, dentre outros. Da mesma forma, em (Di Tria et al., 2017) são propostas métricas para avaliar esquemas multidimensionais, incluindo métricas para avaliar o custo de implementar esquemas para *data warehouses*. Exemplos de critérios e métricas para avaliar esquemas são (Moody, 1998):

- Exatidão: número de violações aos padrões de modelagem de dados, incluindo convenções na diagramação do esquema, regras de nomeação de elementos, regras de composição de entidades e redundância de elementos.
- Completude: número de elementos no esquema que não correspondem aos requisitos do usuário, número de requisitos que não estão representados no esquema, número de elementos do esquema associados aos requisitos, mas especificados incorretamente, porcentagem de consultas passíveis de execução.
- Complexidade: número de entidades e número de relacionamentos identificados no esquema, em que esquemas com número maior de elementos são considerados mais complexos.
- Minimalidade: aplica as métricas de complexidade, na qual esquemas que contêm o mínimo possível de elementos são preferíveis.
- Compreensibilidade: aplica as métricas de complexidade em conjunto com métricas de avaliação por parte do usuário e por parte do desenvolvedor para medir a facilidade de compreensão do esquema.

As métricas empregadas na área de engenharia de *software* e as métricas empregadas para avaliar esquemas XML e esquemas relacionais podem ser adaptadas para avaliar como os dados estão estruturados em bancos de dados NoSQL. Apesar de bancos NoSQL serem livres de esquema, as instâncias de dados têm uma estrutura que precisa ser conhecida pela aplicação para recuperar os dados em momento futuro. Essa estrutura pode ser considerada como uma espécie de esquema e ser utilizada como base para medições e avaliações.

2.5 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Neste capítulo foram apresentados conceitos sobre bancos de dados relacionais e não relacionais (NoSQL), conceitos sobre dados estruturados, semiestruturados e não-estruturados, assim como conceitos de transformação e conversão de dados, correspondência e mapeamento entre esquemas. Todos esses conceitos têm relação direta com a abordagem de conversão de RDB para NoSQL proposta nesta tese.

O modelo de programação *MapReduce* será usado para expressar as transformações de dados entre bancos de dados relacionais e bancos de dados NoSQL. Além disso, foi apresentado o *framework* de processamento de dados distribuído *Apache Spark*, como exemplo de implementação do modelo *MapReduce*. As características do *Apache Spark*, como possibilidade de manipular vários formatos de dados e executar transformações de dados de forma distribuída e paralela foram consideradas para selecioná-lo como *framework* adjacente do processo de migração definido no Capítulo 5.

Por fim, foram apresentados conceitos sobre uso de métricas para avaliar *software*, esquemas XML, esquemas relacionais e multidimensionais. Estes conceitos se relacionam ao conjunto de métricas para avaliar e comparar esquemas NoSQL proposto no Capítulo 7.

No próximo capítulo serão apresentados os trabalhos relacionados ao tema de pesquisa desta tese.

3 ESTADO DA ARTE

Este capítulo apresenta os trabalhos relacionados ao tema de pesquisa. Na primeira seção são apresentadas abordagens de conversão e migração de dados entre bancos de dados relacionais e bancos de dados NoSQL orientados a documentos e famílias de colunas. Na segunda seção são apresentadas abordagens que definem técnicas de modelagem de dados para bancos de dados NoSQL. A terceira seção apresenta abordagens baseadas em métricas para avaliar modelos de dados NoSQL. Por último, são apresentadas as limitações desses trabalhos e discutidas as motivações para o desenvolvimento da abordagem de conversão de RDB para NoSQL e migração de dados desenvolvida nesta tese.

3.1 CONVERSÃO DE RDB PARA NOSQL

De forma geral, a conversão de modelo relacional para NoSQL consiste em três etapas: *(1)* definição de mapeamentos entre conceitos de ambos os modelos; *(2)* definição de regras de transformação; *(3)* execução do processo de conversão. Foram identificadas abordagens considerando essas três etapas. Essas abordagens foram agrupadas de acordo com o modelo de NoSQL suportado, sendo apresentadas a seguir.

3.1.1 NoSQL Orientado a Família de Colunas

(Vajk et al., 2013) apresentaram um algoritmo para gerar esquema de dados desnormalizado para bancos NoSQL orientados a família de colunas. O algoritmo recebe como entrada esquema relacional e conjunto de consultas. As consultas representam a carga de trabalho do banco relacional e devem ser conhecidas antecipadamente. O algoritmo percorre todas as tabelas do RDB e verifica as dependências entre elas. Para cada dependência entre duas tabelas uma terceira tabela desnormalizada é criada. As consultas são transformadas conforme esquema NoSQL gerado, resultando em pares de tabelas e consultas. A opção que apresentar menor custo em termos de armazenamento é selecionada.

(Zhao et al., 2014) propuseram uma metodologia para migração de bancos de dados relacionais para HBase com suporte a múltiplos aninhamentos. O principal objetivo da abordagem é agrupar tabelas relacionadas em uma única tabela HBase e evitar consultas com junções. A metodologia proposta analisa as dependências entre as tabelas do RDB para realizar a conversão, em que uma tabela do RDB gera uma tabela no HBase com uma família de colunas para armazenar seus atributos. Na sequência, as tabelas RDB relacionadas são adicionadas como famílias de colunas da tabela HBase. Também foi proposto um algoritmo para mapear as consultas SQL sobre o esquema HBase gerado.

(Serrano et al., 2015) apresentaram um método heurístico para mapear esquema de dados relacional para esquema HBase. O método tem quatro passos: *(i)* desnormalização sistemática, *(ii)* extensão de junção de tabelas, *(iii)* codificação de chaves e *(iv)* criação de visões materializadas. O primeiro passo analisa as dependências entre tabelas e une elas seguindo algumas heurísticas, de acordo com a cardinalidade dos relacionamentos. Através de um esquema de nomeação de colunas, a abordagem permite aninhar as entidades do lado *N* no mesmo registro da entidade do lado um. O passo de extensão de junção de tabelas procura por tabelas vizinhas relacionadas e tenta uní-las. Esse processo de junção continua até um determinado nível de junções ou enquanto relacionamentos *1:N* forem identificados. Para o terceiro passo os autores

forneceram algumas diretrizes para selecionar a *rowkey* das tabelas HBase. O quarto passo consiste em usar visões materializadas para fornecer caminhos alternativos de acesso aos dados.

(Lee e Zheng, 2015) apresentaram uma abordagem automática para desnormalizar e migrar RDB para HBase. A abordagem usa o princípio DDI (Desnormalização, Duplicação e Chaves Inteligentes), procurando reagrupar as tabelas do RDB e evitar consultas com junção. Os autores apresentaram um algoritmo que lê os metadados do RDB e realiza o processo de desnormalização. Para cada tabela do RDB é criada uma lista vinculada, composta por tabelas relacionadas por meio da análise de dependência entre elas. O processo executa de forma recursiva até que todas as tabelas referenciadas sejam analisadas. O resultado é um conjunto de listas vinculadas representando uma cadeia de relacionamentos. Para cada lista vinculada é determinada a *rowkey* da tabela HBase baseada na concatenação de todas as chaves primárias das tabelas RDB. Os autores não consideraram distribuir os dados em família de colunas.

(Santos e Costa, 2016) propuseram um conjunto de regras para transformação do modelo de dados relacional para modelo de dados orientado a família de colunas (HBase). O objetivo é obter um modelo para análise de dados no contexto de *big data*. A abordagem organiza os dados do RDB em famílias de colunas descritivas e analíticas, em que as famílias descritivas são formadas por tabelas que recebem apenas cardinalidade 1 (um) no modelo relacional e são usadas para complementar a descrição de outras entidades. As demais tabelas do modelo relacional formam as famílias de colunas analíticas, que são definidas através da análise das dependências entre tabelas do banco relacional (chaves primárias e estrangeiras). Após definir as famílias de colunas são definidas as tabelas no banco de dados NoSQL. Uma tabela é definida por uma família descritiva ou por um grupo de famílias analíticas relacionadas. Para definir a *rowkey* da tabela nos bancos de dados NoSQL foi sugerido concatenar as chaves primárias das tabelas RDB relacionadas.

3.1.2 NoSQL Orientado a Documento

(Zhao et al., 2014) apresentaram um modelo de conversão de esquema relacional para NoSQL documento. A abordagem usa o conceito de tabelas aninhadas para evitar consultas com junção. Um algoritmo de transformação baseado em grafo foi proposto, em que tabelas são vértices e dependências são arestas. Também foi proposto um modelo de extensão de tabelas composto por: extensão simples, extensão vertical e extensão horizontal. Basicamente os modelos de extensão definem como analisar as dependências entre as tabelas, com o objetivo de desnormalizar e aninhar os dados usando documentos embutidos ou *arrays* de documentos embutidos. Os autores usaram essa abordagem para migrar dados de MySQL para MongoDB.

(Stanescu et al., 2016) propuseram um algoritmo para automaticamente mapear banco MySQL para MongoDB. Mapeamentos entre os conceitos de ambos os bancos e estratégias para implementar relacionamentos *1:1*, *1:N* e *N:M* foram apresentados. O algoritmo recebe como entrada os metadados do RDB e analisa as dependências entre as tabelas. Seis passos foram propostos para decidir como relacionar as entidades do RDB usando documentos embutidos ou referências. Para tal, o algoritmo considera as cardinalidades dos relacionamentos.

(El Alami e Bahaj, 2016) forneceram uma abordagem para migrar RDB para MongoDB, em que foi proposto um modelo de dados para representar o RDB e um conjunto de regras de conversão de relacionamentos. O modelo de dados representa as tabelas, campos e relacionamentos do RDB, com respectivas cardinalidades. Esse modelo de dados é criado a partir da análise de dependências entre as tabelas do RDB, de forma automática. No próximo passo as regras de transformação são aplicadas sobre o modelo de dados para criar o modelo físico do banco de dados NoSQL. A abordagem não usa desnormalização de dados no processo de transformação,

migrando as tabelas como coleções de documentos independentes e relacionando os documentos através de referências.

(Karnitis e Arnicans, 2015) descreveram uma abordagem para migrar RDB para NoSQL documento. Para tal, um modelo composto por dois níveis lógicos foi proposto. O primeiro nível consiste em um metamodelo para representar as estruturas físicas e lógicas do RDB. O principal componente do metamodelo é chamado de *table-like-structure* (TLS), que representa uma tabela RDB com seus atributos e relacionamentos. A TLS tem quatro subclasses: *codifier*, *simple entity*, *complex entity* e *N:N-link*. Essas classes são usadas para suplementar o modelo físico do RDB, com informações para auxiliar no processo de desnormalização dos dados. Para gerar as TLS os autores usaram um algoritmo proposto em trabalho anterior (Arnicans e Janis, 1998), que faz análise das dependências entre as tabelas do RDB. O segundo nível lógico consiste em uma estrutura em árvore composta por uma TLS como vértice raiz e demais TLS relacionadas. Essa árvore pode ser refinada pelo usuário, que pode remover ou inserir outras TLSs e selecionar quais campos serão migrados. O resultado é um modelo enriquecido com informações usadas para extrair os dados do RDB e migrar para NoSQL documento. Os autores validaram a abordagem por meio de um estudo de caso de migração de MySQL para MongoDB.

Em (Jia et al., 2016) foi proposta uma abordagem de transformação de modelo e migração de dados entre RDB e NoSQL documento. A abordagem leva em consideração as características dos dados e consultas do RDB. Os autores propuseram quatro rótulos para descrever as características do RDB: *(i) Frequent Join*, *(ii) Big Size*, *(iii) Frequent Modify* e *(iv) Frequent Insert*. Esses quatro rótulos são usados para anotar entidades e relacionamentos do RDB, com o objetivo de decidir quando usar documentos embutidos ou referências para representar os relacionamentos do RDB. Os autores apresentaram um algoritmo que recebe o modelo entidade-relacionamento anotado com os rótulos e analisa as dependências entre as entidades para gerar o modelo NoSQL físico. Para tabelas anotadas com os rótulos *Big Size*, *Frequent Modify* ou *Frequent Insert* não são usados documentos embutidos, mas sim referências. No último passo, o modelo físico é utilizado para migrar os dados do RDB para MongoDB.

Mongify[1] é um sistema de tradução de dados para carregar os dados do RDB para MongoDB. Esse sistema permite especificar como que tabelas serão migradas para MongoDB através da definição de *scripts*. É possível selecionar e renomear campos, além de definir se as tuplas de uma tabela serão migradas como documentos ou *array* de documentos embutidos. No entanto, *Mongify* é uma abordagem para transformação e migração de dados entre dois sistemas de armazenamento de dados específicos. Além disso, não tem suporte para outras tecnologias de armazenamento de dados e não apresenta mecanismos de extensão, o que dificulta a adaptação para outros cenários.

3.1.3 Múltiplos Modelos de NoSQL

Em (Freitas et al., 2016) foi proposta uma ferramenta para converter RDB para NoSQL chamada R2NoSQL. Os autores estabeleceram mapeamentos conceituais de RDB para NoSQL orientado a chave-valor, família de colunas, documentos e grafo. A abordagem de conversão é baseada em três passos: *(i)* definição de mapeamentos conceituais entre RDB e um dos tipos de NoSQL; *(ii)* uso dos mapeamentos na conversão de metadados e dados; *(iii)* classificação de tabelas do RDB para auxiliar no processo de desnormalização. Foram propostas quatro classes de tabelas: principal, subclasse, comum e relacionamento. O usuário especialista deve informar manualmente a classificação das tabelas. R2NoSQL recebe como parâmetros o esquema do RDB e classificação das tabelas, e faz análise das dependências entre as tabelas do RDB para

[1] `http://mongify.com/`

decidir como representar os relacionamentos no banco NoSQL alvo (referências ou documentos embutidos). Apesar de fornecer mapeamentos para todas as categorias de NoSQL, a versão do R2NoSQL apresentada suporta apenas MongoDB.

NotaQL é uma linguagem de transformação de dados desenvolvida para ler e transformar dados entre bancos NoSQL (Schildgen et al., 2016). Na versão atual são suportados os bancos MongoDB, HBase, Redis, arquivos CSV (*Comma-Separated-Values*) e JSON. *NotaQL* foi construída para ser extensível, na qual novos bancos NoSQL podem ser adicionados pelo usuário através da implementação de novas classes Java. *NotaQL* usa *Apache Spark* como *framework* de processamento de dados adjacente. Os seguintes tipos de transformações de dados são suportados: renomeação de campos, projeções, filtros e agregações. A gramática da linguagem do *NotaQL* pode ser estendida, para definir expressões de processamento de entrada e saída de dados específicos para cada banco NoSQL suportado. Funções definidas pelo usuário em Java podem ser adicionadas ao *NotaQL*, permitindo estender as funcionalidades da linguagem. Apesar do suporte a arquivos CSV, *NotaQL* não tem suporte para bancos de dados relacionais.

Transporter[2] é uma ferramenta de código aberto para transformar dados entre diferentes sistemas de armazenamento. A ferramenta tem um sistema simples de operação, em que o usuário precisa definir as fontes de origem e destino. Funções de transformação podem ser adicionadas entre as fontes de origem e destino, definindo uma espécie de *pipeline* de transformações. A partir da configuração da fonte de origem, os dados são extraídos e convertidos para mensagens em formato JSON e enviados para a fonte de destino. Dois tipos de transformações são suportados: nativas e Javascript. As transformações definidas em Javascript possibilitam estender a ferramenta com novas operações lógicas sobre os dados. As transformações nativas permitem executar projeções, filtros, renomeação de campos e impressão de dados no console. No entanto, não tem suporte para agrupamentos de dados e processamento distribuído. Além disso, conforme documentação do *Transporter*, as transformações nativas sobre dados semiestruturados operam apenas sobre campos de nível superior, sem suporte para campos aninhados. Na versão atual, a ferramenta tem suporte aos seguintes sistemas de armazenamento: Elasticsearch, MongoDB, PostgreSQL, RethinkDB e arquivos.

Em (Alotaibi e Pardede, 2019) foram definidas regras para transformação de esquema relacional para bancos de dados NoSQL orientados a famílias de colunas, documentos e grafos. As regras propostas consideram as cardinalidades entre as entidades do RDB e cobrem os relacionamentos de associação, herança e agregação. Não foi definida ferramenta para implementar as regras de transformação e realizar a migração dos dados. Trata-se de uma abordagem manual, conduzida pelo usuário especialista.

3.2 MODELAGEM DE DADOS PARA NOSQL

Diferentes abordagens apresentaram definições formais de esquemas NoSQL. Em (Bugiotti et al., 2014), os autores apresentaram NoAM (*NoSQL Abstract Model*), que usa como principal unidade de modelagem o conceito de agregados (conjunto de entidades que são acessadas juntas) e é dirigido pelos casos de uso da aplicação (requisitos funcionais). O usuário especialista modela os agregados de dados a partir dos dados da aplicação e padrão de acesso desejado. O resultado é uma representação intermediária, independente de sistema, em que no último passo deve ser traduzida para o modelo NoSQL alvo.

Em (Xiang Li et al., 2014) os autores apresentaram uma abordagem de modelagem orientada a consultas (QODM, do inglês *Query-Oriented Data Modeling*), em que o usuário especialista define o modelo de dados e consultas da aplicação. Com base nas consultas a

[2] `https://github.com/compose/transporter`

abordagem define quais entidades serão aninhadas/desnormalizadas no modelo NoSQL gerado. As consultas são representadas por meio de tuplas no formato $< C_k, C_t >$, em que C_k relaciona o conjunto de entidades que representa o filtro da consulta e C_t o conjunto de entidades que representa o resultado da consulta. Com base na análise das tuplas (consultas) é gerado um modelo de dados com entidades agregadas, que são entidades produzidas pela desnormalização das entidades relacionadas pela consulta. Esse modelo de dados é usado para definir o esquema de dados para bancos de dados NoSQL. Apesar da abordagem considerar o padrão de acesso da aplicação para definir um esquema de dados NoSQL, não são consideradas a ordem de aninhamento entre as entidades e nem a frequência de execução das consultas durante o processo.

Em (Mior et al., 2016) foi proposto NoSE, uma ferramenta para recomendar esquemas para bases NoSQL orientadas a famílias de colunas. NoSE usa um modelo de custo para recomendar famílias de colunas, com base em um modelo conceitual dos dados que serão manipulados pela aplicação e um conjunto de consultas que representa a carga de trabalho requerida. A ferramenta gera como saída um conjunto de famílias de colunas que cobre toda a carga de trabalho e planos de execução das consultas. O desenvolvedor usa esquema e os planos de consultas como subsídio para desenvolver a aplicação sobre a base NoSQL. Apesar de considerar as consultas da aplicação para gerar um esquema NoSQL, NoSE não é destinado para migrar bases relacionais para bases NoSQL. A abordagem tem como alvo apenas bases NoSQL orientadas a família de colunas e não apresenta diferentes formas de aninhamento dos dados, como por exemplo, aninhamentos *1:N*, em que o lado muitos é aninhado dentro do registro do lado um.

(Abdelhedi et al., 2017) apresentaram uma abordagem baseada em MDA (do inglês, *Model-Driven Architecture*) para automaticamente transformar um modelo em UML (do inglês, *Unified Modeling Language*) para modelo físico NoSQL. Um conjunto de regras de transformação são apresentados para gerar o modelo físico. São suportados bancos de dados NoSQL orientados a documentos, famílias de colunas e grafos.

3.3 MÉTRICAS PARA AVALIAR ESQUEMAS NOSQL

Considerando a utilização de métricas para avaliar esquemas NoSQL, o trabalho de (Gómez et al., 2018) apresentou um conjunto de métricas estruturais para avaliar esquemas de bancos de dados NoSQL orientados a documentos. Uma abstração baseada em grafo é utilizada para representar o esquema NoSQL e permitir a computação das métricas. Esse trabalho foi baseado em (Pusnik et al., 2014) e (Klettke et al., 2002), que apresentaram métricas estruturais para avaliar documentos XML. As métricas propostas são apresentadas abaixo, em que c representa uma determinada coleção de documentos do esquema e t_{doc} o tipo do documento:

- $colExistence(t_{doc})$: determina a existência de uma coleção que armazena documentos do tipo t_{doc}, localizados no primeiro nível da coleção.
- $docExistence(t_{doc}, c)$: determina a existência de documentos do tipo t_{doc} na coleção c.
- $colDepth(c)$: retorna a profundidade máxima da coleção c.
- $globalDepth()$: retorna a profundidade máxima do esquema, aplicando $colDepth(c_i)$ para todas as coleções (c_i) do esquema.
- $docDepthInCol(c, t_{doc})$: retorna a profundidade do documento t_{doc} na coleção c.
- $maxDocDepth(t_{doc})$: retorna a profundidade máxima em que o documento do tipo t_{doc} é localizado no esquema.

- $minDocDepth(t_{doc})$: retorna a profundidade mínima em que o documento do tipo t_{doc} é localizado no esquema.
- $docWith(c, t_{doc})$: retorna o tamanho do documento do tipo t_{doc} na coleção c. Essa métrica usa pesos distintos para calcular o tamanho do documento, de acordo com os tipos de atributos encontrados (tipo primitivo, *array* de tipo primitivo, documento embutido, *array* de documento embutido).
- $refLoad(c)$: número de atributos de outros documentos que potencialmente referenciam os documentos da coleção c.
- $docCopiesInCol(c, t_{doc})$: número de cópias do documento do tipo t_{doc} na coleção c.
- $docTypeCopies(t_{doc})$: retorna o número de vezes que um documento do tipo t_{doc} é usado no esquema.

Essas métricas focam em aspectos relacionados à complexidade estrutural do esquema, e têm por objetivo auxiliar o usuário no processo de seleção do esquema mais apropriados aos requisitos da aplicação. No entanto, o trabalho não apresenta métricas específicas para avaliar o padrão de acesso da aplicação em relação ao esquema NoSQL, que é um aspecto chave no processo de seleção do esquema de dados.

3.4 DISCUSSÕES SOBRE O ESTADO DA ARTE

Esta seção apresenta uma análise dos trabalhos relacionados e aspectos que podem ser explorados, que motivaram a pesquisa realizada nesta tese.

Em relação à conversão para NoSQL orientado a família de colunas, com exceção da abordagem (Zhao et al., 2014), todas as demais (Santos e Costa, 2016; Vajk et al., 2013; Serrano et al., 2015; Lee e Zheng, 2015) são automáticas e apresentam algoritmos que fazem análise de dependências entre as tabelas do RDB e executam desnormalização de dados. Além disso, não oferecem suporte para customizações e extensões no processo de conversão. A customização se refere ao suporte para operações como seleção de campos, tabelas, filtro de instâncias ou meios de personalizar o modelo de dados de saída antes de realizar a migração de dados.

Em relação à conversão para NoSQL orientado a documentos, as abordagens (Stanescu et al., 2016; Zhao et al., 2014; El Alami e Bahaj, 2016) são automáticas e sem forma direta de customizar ou estender o processo de conversão. As abordagens (Freitas et al., 2016; Karnitis e Arnicans, 2015; Jia et al., 2016) são semi-automáticas, em que são analisadas as dependências entre as tabelas do RDB, em conjunto com a classificação de tabelas fornecida pelo usuário especialista. O principal objetivo da classificação das tabelas é auxiliar na decisão de usar documentos embutidos ou referências no momento de converter os relacionamentos do RDB para NoSQL.

Mongify foi desenvolvida para transformar e migrar dados entre RDB e MongoDB, sem forma direta de estender para outros bancos de dados. *Transporter* e *NotaQL* são ferramentas para transformar dados semiestruturados, por meio de funções definidas pelo usuário em *Javascript*. Porém, *Transporter* não tem suporte para agrupamento de dados e processamento distribuído e *NotaQL* (Schildgen et al., 2016) não fornece suporte a bancos de dados relacionais, apesar de ser uma linguagem de transformação de dados extensível. Em (Alotaibi e Pardede, 2019) foi apresentada uma abordagem manual, em que o usuário aplica as regras de transformação para migrar RDB para NoSQL.

A Tabela 3.1 apresenta um comparativo entre as abordagens de conversão investigadas. Os seguintes atributos são apresentados:

- **Autores**: autores e ano do trabalho.
- **Modelo (E)**: modelo de dados de entrada da abordagem (relacional, orientado a documentos (Doc), orientado a família de colunas (CF) e/ou orientado a grafo (*Graph*)).
- **Modelo (S)**: modelo de dados de saída gerado pela execução da abordagem (considerar as mesmas siglas do item anterior).
- **Entradas**: entradas requeridas pela abordagem, como metadados do RDB, consultas e outras informações fornecidas pelo usuário.
- **Características**: informa quais são as principais características da abordagem, empregadas no processo de conversão de RDB para NoSQL.
- **Customização (Custom.)**: informa se a abordagem tem suporte para customização do processo de conversão, como seleção de campos, tabelas, filtro de instâncias ou meios de customizar o modelo de dados de saída, antes de realizar a migração de dados.
- **Automação**: informa se a abordagem é automática (auto), semi-automática (semi) ou manual.
- **DB Destino**: informa quais são os bancos de dados NoSQL suportados pela abordagem.
- **Avaliação**: informa se a abordagem tem suporte a uma fase de avaliação do modelo de dados de saída. Critérios de avaliação aplicáveis são: o tamanho da base de dados, a redundância de dados, a adequação ao padrão de acesso da aplicação, dentre outros.

Nesta tese, será apresentado um *framework* de transformação de dados extensível capaz de executar transformações de dados estruturados e semiestruturados, através de funções *MapReduce*, possibilitando a execução de transformações de forma paralela e distribuída. O *framework* tem suporte para transformações de RDB para NoSQL orientado a documentos e orientado a família de colunas. Um dos objetivos do *framework* é ser genérico o suficiente para permitir expressar as regras de conversão das abordagens apresentadas neste capítulo.

Outro aspecto importante no processo de conversão é a definição do modelo de dados (ou esquema NoSQL) adequado aos requisitos da aplicação. A escolha do esquema NoSQL influencia em vários fatores, como o tempo de execução de consultas, a redundância de dados, o esforço de implementação das consultas e, também, na manutenção da aplicação. Este capítulo apresentou abordagens relacionadas a modelagem de dados para bancos de dados NoSQL (Bugiotti et al., 2014; Xiang Li et al., 2014; Mior et al., 2016; Abdelhedi et al., 2017). Porém, essas abordagens não fornecem meios específicos para avaliar e comparar os possíveis modelos de dados NoSQL gerados em relação aos requisitos da aplicação.

Uma alternativa é o uso de métricas para avaliar e comparar os esquemas candidatos antes de realizar a conversão e migração dos dados. O trabalho (Gómez et al., 2018) apresentou onze métricas estruturais para avaliar esquemas NoSQL orientados a documentos. De forma diferente, nesta tese será apresentado um conjunto de métricas para avaliar e comparar diferentes opções de esquemas NoSQL orientados a documentos em relação ao padrão de acesso da aplicação. Apesar da abordagem definida neste trabalho ser parcialmente inspirada no trabalho (Gómez et al., 2018), em que ambos usam uma abstração semelhante para representar esquemas NoSQL, o foco é diferente, voltado ao padrão de acesso da aplicação em vez de avaliar somente aspectos estruturais dos esquemas.

Tabela 3.1: Resumo das abordagens de conversão de RDB para NoSQL. As abordagens foram ordenadas por Modelo(S), Características e Ano de Publicação.

Autores	**Modelo (E)**	**Modelo (S)**	**Entradas**	**Características**	**Custom.**	**Automação**	**DB Destino**	**Avaliação**
(Vajk et al., 2013)	Relacional	CF	Metadados e SQL	Análise de dependências entre tabelas. Uso de desnormalização.	Não	Auto	N/A	Não
(Zhao et al., 2014)	Relacional	CF	Metadados e SQL	Análise de dependências entre tabelas. Uso de desnormalização.	Não	Auto	HBase	Não
(Serrano et al., 2015)	Relacional	CF	Metadados	Análise de dependências entre tabelas. Uso de desnormalização.	Não	Auto	HBase	Não
(Lee e Zheng, 2015)	Relacional	CF	Metadados	Análise de dependências entre tabelas. Uso de desnormalização.	Não	Auto	HBase	Não
(Santos e Costa, 2016)	Relacional	CF	Metadados	Análise de dependências entre tabelas. Uso de desnormalização.	Não	Auto	HBase	Não
(Zhao et al., 2014)	Relacional	Doc	Metadados.	Análise de dependências entre tabelas. Uso de desnormalização.	Não	Auto	MongoDB	Não
(Stanescu et al., 2016)	Relacional	Doc	Metadados.	Análise de dependências entre tabelas. Uso de referências e desnormalização.	Não	Auto	MongoDB	Não
(El Alami e Bahaj, 2016)	Relacional	Doc	Metadados.	Análise de dependências entre tabelas. Uso de desnormalização.	Não	Auto	MongoDB	Não
(Karnitis e Arnicans, 2015)	Relacional	Doc	Metadados. Classificação de tabelas.	Análise de dependências entre tabelas. Análise da classificação de tabelas. Uso de referências, desnormalização.	Sim	Semi	MongoDB	Não
(Jia et al., 2016)	Relacional	Doc	Metadados. Classificação de: - Tabelas e Relacionamentos.	Análise de dependências entre tabelas. Análise de classificação de tabelas. Uso de referências e desnormalização.	Sim	Semi	MongoDB	Não
Mongify	Relacional	Doc	*Scripts* do usuário	Operadores de transformação de dados.	Sim	Semi	MongoDB	Não
(Freitas et al., 2016)	Relacional	CF/Doc/Graph	Metadados Classificação de tabelas.	Análise de dependências entre tabelas. Análise da classificação de tabelas. Uso de referências e desnormalização.	Sim	Semi	MongoDB	Não
(Schildgen et al., 2016)	CF/Doc/Graph	CF/Doc/Graph	*Scripts* do usuário	Operadores de transformação de dados.	Sim	Semi	MongoDB HBase Neo4j	Não
Transporter	Relacional CF/Doc	CF/Doc Relational	*Scripts* do usuário	Operadores de transformação de dados.	Sim	Semi	MongoDB ScyllaDB PostgreSQL	Não
(Alotaibi e Pardede, 2019)	Relacional	CF/Doc/Graph	Metadados	Análise de dependências entre tabelas. Uso de referências e desnormalização.	Não	Manual	N/A	Não

3.5 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Neste capítulo foram apresentados trabalhos relacionados ao contexto de conversão e migração de bases de dados relacionais para bancos de dados NoSQL, bem como suas principais características e limitações. No próximo capítulo será apresentada a abordagem de conversão de bancos de dados relacionais para NoSQL e migração de dados definida nesta tese.

4 ABORDAGEM DE CONVERSÃO DE RDB PARA NOSQL E MIGRAÇÃO DE DADOS

Devido à flexibilidade que os bancos de dados NoSQL fornecem em termos de estruturação dos dados, a escolha do formato adequado (ou esquema) não é uma tarefa trivial. A escolha depende de vários aspectos, como o padrão de acesso da aplicação, o nível de redundância de dados desejado/aceitável, o tamanho da base de dados, o esforço de manutenção da aplicação, entre outros. Diferentes abordagens para converter bases de dados relacionais para bases NoSQL foram propostas na literatura, conforme consta no levantamento bibliográfico realizado. De forma geral, essas abordagens desnormalizam os dados do RDB analisando as dependências entre tabelas e/ou padrão de acesso aos dados. No entanto, nenhuma dessas abordagens preocupa-se ou têm uma fase específica para avaliar se a estruturação dos dados gerada é adequada aos requisitos da aplicação. Com base nisso, o propósito desta tese é apresentar uma abordagem de conversão de RDB para NoSQL que aborde essa problemática.

Neste capítulo são apresentadas as etapas e a arquitetura da abordagem de conversão proposta, destacando seus principais componentes. A arquitetura foi construída para atingir o objetivo geral desta tese, que *é desenvolver uma abordagem de conversão de bases relacionais para bases NoSQL orientadas a documentos e famílias de colunas, com suporte à avaliação e comparação de esquemas NoSQL candidatos em relação ao padrão de acesso da aplicação*. A abordagem é composta por quatro etapas, em que cada uma delas aborda um ou mais dos objetivos específicos definidos neste trabalho.

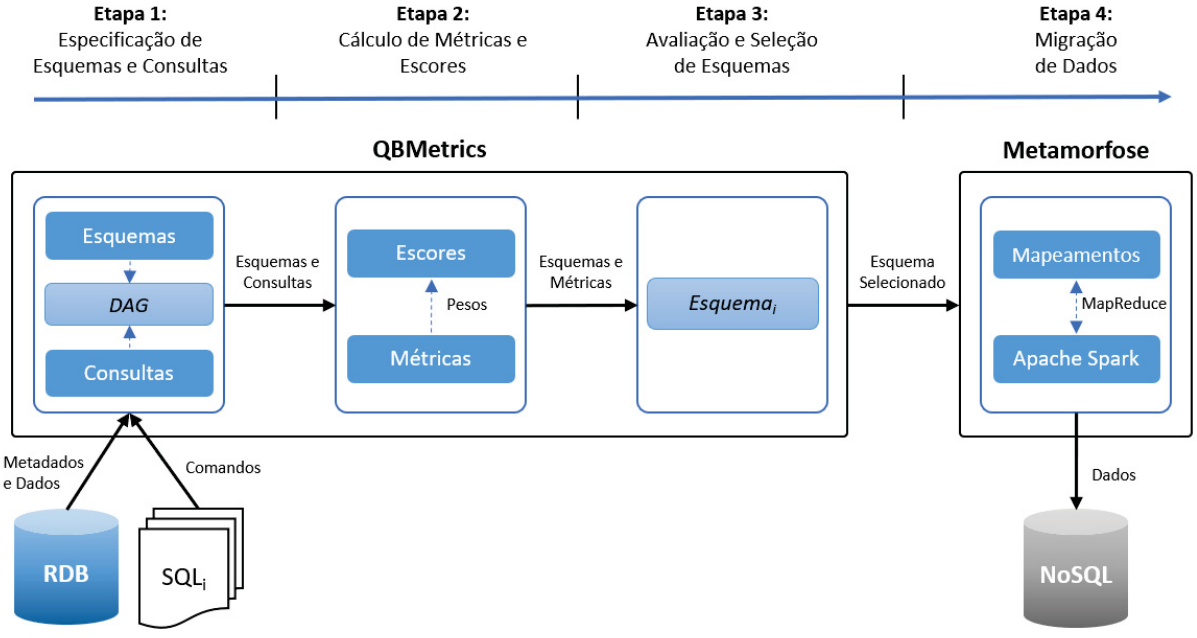


Figura 4.1: Etapas e arquitetura da abordagem de conversão de RDB para NoSQL e migração de dados.

A Figura 4.1 exibe as etapas e a arquitetura da abordagem. Na etapa **1. Especificação de Esquemas e Consultas** o usuário define manualmente um conjunto de esquemas candidatos e consultas a partir do RDB de entrada. Esquemas e consultas são representados por meio de grafos acíclicos direcionados (DAGs). Na etapa **2. Cálculo de Métricas e Escores** é calculada a cobertura fornecida pelos esquemas definidos na etapa anterior em relação ao padrão de acesso das consultas de entrada. Na etapa **3. Avaliação e Seleção de Esquemas** o usuário realiza a avaliação dos esquemas NoSQL candidatos por meio das métricas e escores calculados na etapa

2, e seleciona um deles para realizar a migração dos dados. As etapas 1 a 3 são realizadas por meio da ferramenta *QBMetrics*, apresentada no Capítulo 7. Por fim, na etapa **4. Migração de Dados** o esquema selecionado é enviado ao Metamorfose, um *framework* de transformação de dados apresentado no Capítulo 5, usado para migrar os dados do RDB para NoSQL.

Os componentes e etapas definidos na arquitetura são introduzidos neste capítulo e apresentados em maiores detalhes em capítulos posteriores.

4.1 ETAPA 1: ESPECIFICAÇÃO DE ESQUEMAS E CONSULTAS

Nesta etapa são definidos um ou mais esquemas NoSQL candidatos para migração de dados e um conjunto de consultas que representa o padrão de acesso da aplicação. O uso de DAGs para representar esquemas e consultas tem por objetivo evitar a ocorrência de ciclos ao estabelecer relacionamentos entre entidades, requisito para o processo de conversão e migração de RDB para NoSQL definido na etapa 4. Ademais, considerando o contexto de conversão de RDB para NoSQL, as consultas são previamente conhecidas e definidas como comandos SQL. As seções a seguir apresentam maiores detalhes sobre a definição de esquemas e consultas.

4.1.1 Esquema NoSQL representado por meio de DAGs

No contexto deste trabalho, DAGs são usados como abstração para representar esquemas de bases NoSQL orientadas a documentos e orientadas a famílias de colunas. Um esquema NoSQL é definido como um conjunto de entidades representadas como DAGs. Uma entidade representada como um DAG tem estrutura de uma árvore e relaciona uma ou mais tabelas do RDB de entrada. Um DAG é definido como $G = (V, E)$, em que o conjunto de vértices V representa as tabelas do RDB e o conjunto de arestas E os relacionamentos entre tabelas. Os vértices encapsulam os metadados da respectiva tabela RDB, incluindo nome da tabela, campos e chave primária. As arestas encapsulam os metadados do relacionamento entre duas tabelas, incluindo as chaves primárias e estrangeiras e qual tabela está no lado um ou lado muitos do relacionamento.

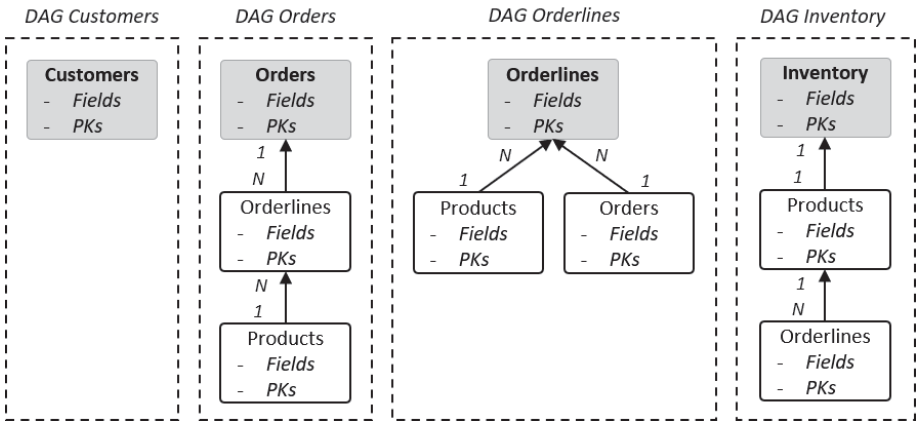


Figura 4.2: Conjunto de DAGs usado para representar entidades NoSQL.

A Figura 4.2 mostra quatro DAGs: *Customers*, *Orders*, *Orderlines* e *Inventory*. O vértice raiz (cor de fundo cinza) representa a entidade NoSQL destino, os demais vértices (cor de fundo branco) representam as entidades aninhadas e a direção da aresta indica a direção do aninhamento. Por exemplo, no DAG *Orders*, *Products* é aninhado a *Orderlines* que por sua vez é aninhado a *Orders*. A estrutura do DAG é usada no processo de conversão e migração dos dados

do RDB para NoSQL, apresentada em detalhes no Capítulo 5. A entidade NoSQL destino pode ser materializada como uma coleção de documentos ou como uma tabela composta por famílias de colunas, sendo dependente do tipo de NoSQL destino. A seguir é apresentado o uso de DAGs para definir esquemas NoSQL orientados a documentos e famílias de colunas.

A Figura 4.3 exibe um esquema NoSQL orientado a documentos com base nos DAGs *Customers*, *Orders*, *Orderlines* e *Inventory*. Cada DAG é mapeado para uma coleção de documentos, em que o vértice raiz representa o primeiro nível da coleção e os demais vértices representam os documentos aninhados (os campos dos documentos não são exibidos na figura). A direção e cardinalidade das arestas definem a direção e tipo do aninhamento entre documentos. Os tipos de aninhamentos são: documento embutido (objeto) quando o relacionamento é *1:1* ou *N:1*, e *array* de documentos embutidos (*array*) quando o relacionamento é *1:N* ou *M:N*. Desta forma, um esquema NoSQL orientado a documentos é definido como $S_{doc} = \{DAG(c)|c \in C\}$, tal que C é um conjunto de coleções de documentos.

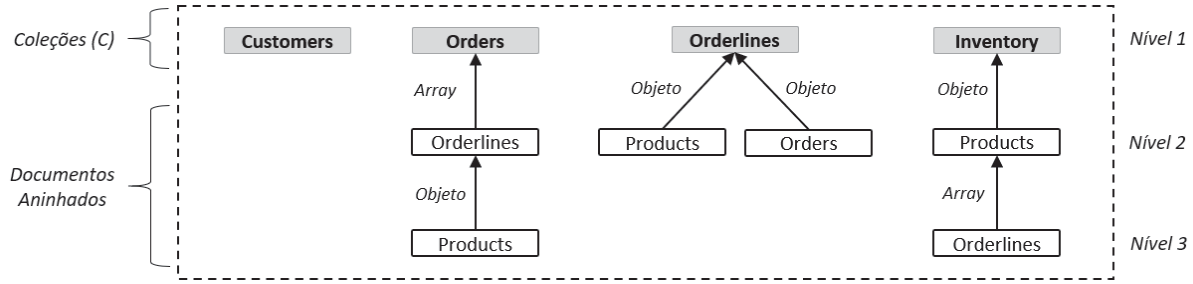


Figura 4.3: Esquema NoSQL orientado a documentos composto por quatro coleções de documentos.

A Figura 4.4 exibe um esquema NoSQL orientado a família de colunas com base nos DAGs *Customers*, *Orders*, *Orderlines* e *Inventory*. Cada DAG é mapeado para uma tabela composta por famílias de colunas. O vértice raiz representa uma tabela e os demais vértices representam as famílias de colunas associadas (os campos das famílias de colunas não são exibidos na figura). A forma do aninhamento por meio de família de colunas é dependente da cardinalidade dos relacionamentos definidos no DAG. Desta forma, um esquema NoSQL orientado a família de colunas é definido como $S_{cf} = \{DAG(tb)|tb \in TB\}$, tal que TB é o conjunto de tabelas do esquema.

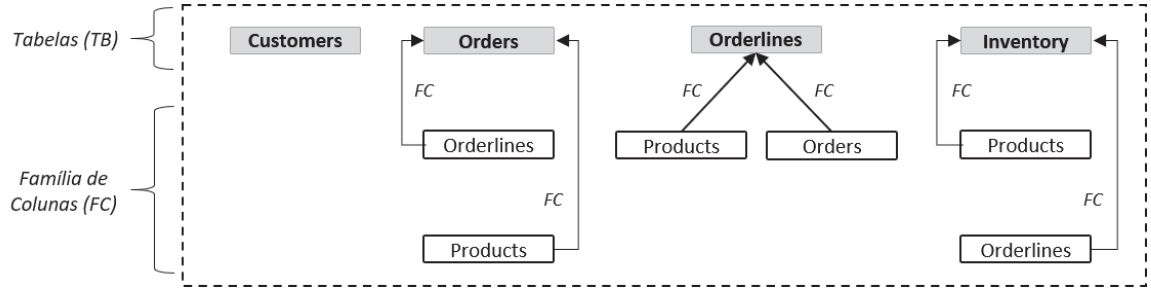


Figura 4.4: Esquema NoSQL orientado a família de colunas composto por quatro tabelas e respectivas famílias de colunas (CF).

4.1.2 Consulta representada por meio de DAG

No contexto deste trabalho são consideradas apenas consultas de leitura. As consultas são representadas como DAGs, denotando o padrão de acesso aos dados, ou seja, o conjunto de tabelas e relacionamentos da consulta. O conjunto de consultas é definido como Q, em que uma

consulta $q \in Q$ é definida como $q = (V_q, E_q)$, tal que V_q é o conjunto de vértices, representando as tabelas da consulta, e E_q é um conjunto de arestas, representando as condições de junção entre as tabelas da consulta.

A abordagem tem suporte apenas para consultas de seleção, sem suporte para sub-consultas, união, agregações e *store procedures*. Planeja-se estender o suporte a mais comandos SQL em trabalhos futuros. Foram definidas duas regras para converter um comando *SELECT* em um DAG. As regras são necessárias para estabelecer um mecanismo de tradução uniforme para representar as consultas como DAGs e realizar a avaliação dos esquemas por meio das métricas apresentadas no Capítulo 7.

- **Regra 1**: se o comando tem somente uma tabela, então um DAG com somente um vértice representando a tabela é criado.
- **Regra 2**: se o comando tem duas ou mais tabelas, então é necessário definir qual tabela é o vértice raiz do DAG. As demais tabelas são adicionadas abaixo do vértice raiz de acordo com as condições de junção do comando SQL.

Para definir o vértice raiz da Regra 2, o comando SQL é analisado (*parsing*) e uma das seguintes sub regras é aplicada:

- **Regra 2.1**: se é um *left join*, retorna a tabela mais à esquerda da cláusula *FROM*.
- **Regra 2.2**: se é um *right join*, retorna a tabela mais à direita da cláusula *FROM*.
- **Regra 2.3**: se é um *inner join*, retorna a primeira tabela a partir da cláusula *FROM*.

Para ilustrar o processo de conversão considere o quadro abaixo, em que são exibidas consultas SQL.

```
1  /* Consulta q1 */
2  select * from orders;
3  /* Consulta q2 */
4  select * from orders left join orderlines on id_order = orderid left join products on id_prod = prod_id;
5  /* Consulta q3 */
6  select * from products left join orderlines on products.id_prod = orderlines.prod_id;
7  /* Consulta q4 */
8  select * from inventory inner join orderlines on inventory.prod_id = orderlines.prod_id;
9  /* Consulta q5 */
10 select * from orders right join customers on id_customer = customerid;
11 /* Consulta q6 */
12 select * from orders o left join customers c on o.customerid = c.id_customer
13     left join orderlines ol on ol.orderid = o.id_order;
```

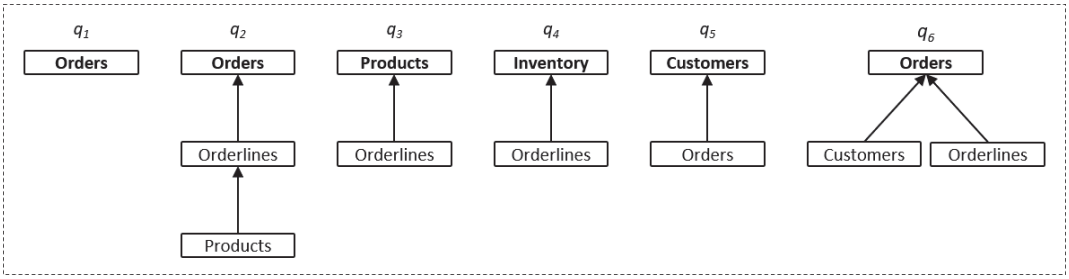


Figura 4.5: Conjunto de consultas SQL representadas como DAGs.

Após aplicar as regras acima sobre as consultas SQL é gerado o conjunto de DAGs exibido na Figura 4.5. Como resultado desta etapa são definidos esquemas e consultas como DAGs. Ambos são usados na próxima etapa para cálculo das métricas e escores.

4.2 ETAPA 2: CÁLCULO DE MÉTRICAS E ESCORES

Esta etapa recebe o conjunto de esquemas NoSQL candidatos e o conjunto de consultas definido na etapa anterior. Foram definidas seis métricas para medir a cobertura que um esquema NoSQL fornece para o conjunto de consultas da aplicação. Como esquemas e consultas são definidos por meio de DAGs, é possível medir a cobertura de forma objetiva. Além das métricas foram definidos dois escores usados para avaliar e comparar esquemas. O primeiro define um escore por consulta e é chamado de $QScore$ (do inglês *Query Score*). Ele permite combinar uma ou mais métricas relacionadas e definir pesos para priorizar a importância de métricas específicas. O segundo é chamado de $SScore$ (do inglês *Schema Score*) e define um escore considerando a cobertura do esquema para todas as consultas, por métrica. A definição das métricas e escores será apresentada no Capítulo 7 em maiores detalhes.

É importante mencionar que a avaliação baseada em métricas definida nesta tese foi aplicada sobre esquemas NoSQL orientados a documentos. Apesar da abordagem permitir converter RDB para NoSQL orientado a família de colunas, o uso das métricas para o modelo orientado a família de colunas será realizada em trabalhos futuros.

4.3 ETAPA 3: AVALIAÇÃO E SELEÇÃO DE ESQUEMA

Nesta etapa o usuário realiza a análise e comparação de esquemas. Os resultados das métricas e escores são usados para identificar quais consultas são cobertas e quais não são cobertas pelo esquema, quais consultas necessitam de duas ou mais coleções de documentos para produzir um resultado, ou para calcular a cobertura que o esquema fornece considerando todas as consultas. O usuário pode definir diferentes pesos para as métricas e consultas, priorizando certo tipo de padrão de acesso ou priorizando certas consultas (consultas executadas frequentemente). No Capítulo 7 serão apresentados maiores detalhes do procedimento para avaliar e comparar esquemas NoSQL candidatos por meio das métricas e escores.

A ferramenta *QBMetrics* fornece um conjunto de relatórios em que o usuário pode visualizar os valores das métricas e escores por esquema, por consulta e por métrica. O objetivo é dar suporte ao usuário no processo de avaliação e seleção do esquema NoSQL mais adequado aos requisitos da aplicação.

4.4 ETAPA 4: MIGRAÇÃO DE DADOS

A última etapa é a migração dos dados do RDB para NoSQL. O esquema NoSQL selecionado na etapa anterior é usado como entrada para o processo de migração de dados. Por meio da ferramenta *QBMetrics* o esquema selecionado é exportado para ser usado pelo Metamorfose, um *framework* de transformação de dados apresentado no Capítulo 5. O esquema exportado encapsula informações de conexão do RDB de entrada, incluindo o endereço do servidor de banco de dados, nome de usuário, senha e nome do banco de dados.

O Metamorfose converte o esquema NoSQL selecionado em comandos para ler os dados do RDB, transformar e persistir em formato NoSQL orientado a documentos ou orientado a família de colunas. Esses comandos encapsulam mapeamentos entre os campos dos esquemas de origem e destino, incluindo funções de transformação de dados. Por fim, o Metamorfose executa os mapeamentos por meio de funções *map* e *mapreduce* implementadas no *Apache Spark*.

4.5 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Neste capítulo foi apresentada a abordagem de conversão de RDB para NoSQL e migração de dados definida nesta tese. Os próximos capítulos apresentam de forma detalhada cada uma das etapas da abordagem e componentes da arquitetura proposta, de forma *bottom up*, iniciando pela etapa de migração de dados.

5 *FRAMEWORK* METAMORFOSE PARA CONVERSÃO E MIGRAÇÃO DE DADOS

Neste capítulo é apresentado em detalhes o processo de conversão e migração de dados de RDB para NoSQL[1], referente à etapa **4. Migração de Dados** definida no Capítulo 4. Os modelos de dados destino são bases NoSQL orientadas a documento e orientadas a família de colunas. De forma geral, a transformação de dados entre bases relacionais e bases NoSQL desnormalizam um conjunto de tabelas do RDB para criar uma entidade NoSQL. Tal entidade NoSQL pode ser representada como uma coleção de documento ou como uma tabela composta por famílias de colunas. Neste trabalho, o processo de desnormalização é representado por meio de um DAG (*Directed Acyclic Graph*).

A Figura 5.1 fornece uma visão geral do processo de conversão e migração de dados, composto pelos seguintes componentes: *(i) Gerador de Comandos*, *(ii) Executor de Comandos* e *(iii) Metamorfose Framework*. O processo recebe um esquema NoSQL, em que cada entidade é representada por um DAG. Na sequência, o componente *Gerador de Comandos* lê cada DAG e gera um conjunto de comandos de transformação de dados. Por último, o componente *Executor de Comandos* invoca operações do *Metamorfose Framework* para ler dados do RDB, transformar e persistir os dados na base NoSQL (formato JSON).

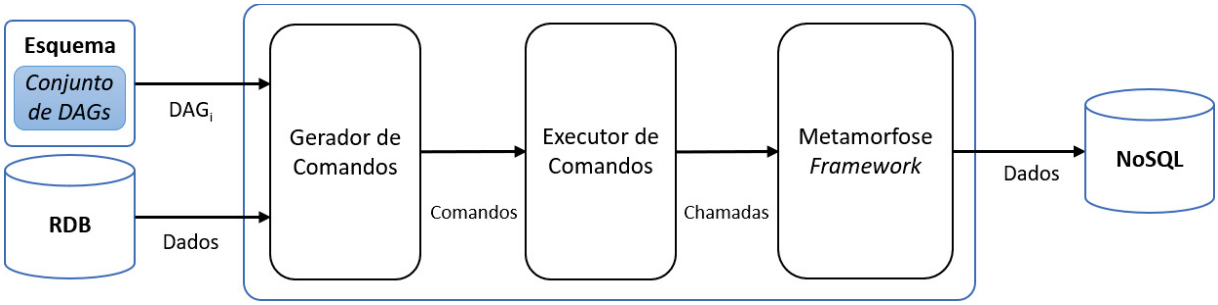


Figura 5.1: Visão geral do processo de conversão e migração de dados de RDB para NoSQL.

No processo de conversão, o uso de DAG tem dois propósitos. O primeiro é estabelecer um caminho de dados, em que esse caminho pode ser usado para representar o processo de desnormalização de tabelas RDB. O segundo propósito é servir como modelo para gerar os mapeamentos e funções de transformação de dados usados pelo Metamorfose para converter instâncias do RDB em instâncias NoSQL.

A seguir serão apresentados detalhes de cada componente da abordagem de conversão. O *framework* Metamorfose será o primeiro componente a ser apresentado, em função de fornecer as primitivas básicas (definição de mapeamentos) para o mecanismo de tradução de DAGs em comandos de transformação de dados.

[1]Parte do conteúdo deste capítulo foi publicado em:

- Kuszera, E. M., Peres, L. M. e Fabro, M. D. D. (2018). Metamorfose: a Data Transformation Framework Based on Apache Spark. Em 33rd Annual Brazilian Symposium on Databases: Proceeedings Companion, SBBD 2018 Companion, Rio de Janeiro, RJ, Brazil, August 25-26, 2018., páginas 11–16.
- Kuszera, E. M., Peres, L. M. e Fabro, M. D. D. (2019). Toward RDB to NoSQL: Transforming Data with Metamorfose Framework. Em Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19, página 456–463, New York, NY, USA. Association for Computing Machinery.

5.1 *FRAMEWORK* METAMORFOSE

O Metamorfose é um *framework* de transformação de dados desenvolvido para dar suporte à abordagem de conversão e migração de RDB para NoSQL. O Metamorfose permite carregar os dados, definir mapeamentos entre campos de origem e destino, executar transformações e persistir os dados. Os mapeamentos encapsulam a lógica de transformação de dados por meio de funções definidas pelo usuário (FDU) em Java ou Javascript. A transformação de dados é executada por meio de funções *map* e *mapreduce*. Esse mecanismo permite transformar dados relacionais para dados semiestruturados (JSON, por exemplo). Ademais, os mapeamentos produzidos podem ser reutilizados e compartilhados entre usuários. A versão atual do Metamorfose suporta arquivos CSV e JSON, e fontes de dados JDBC.

5.1.1 Arquitetura do Metamorfose

A Figura 5.2 apresenta os principais componentes do Metamorfose.

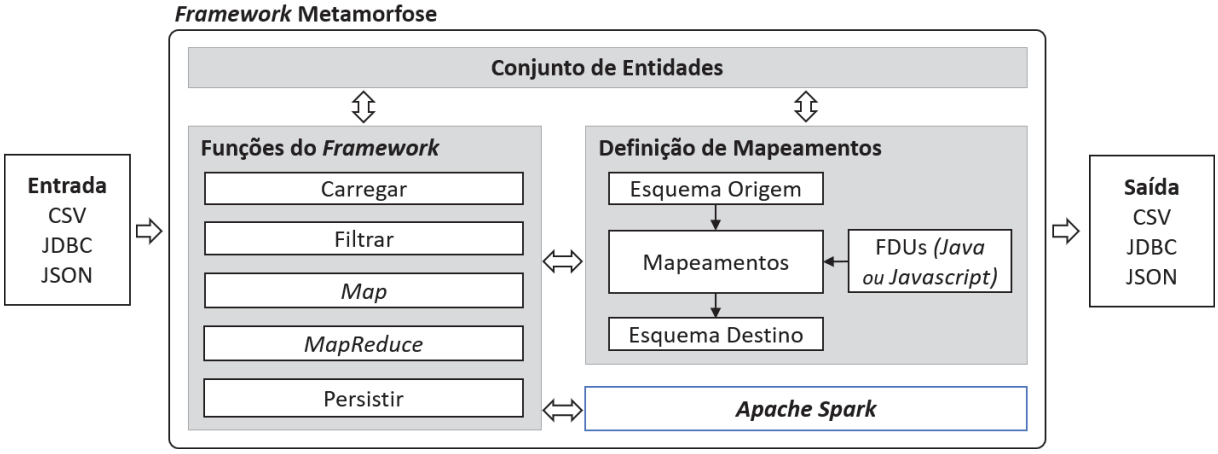


Figura 5.2: Arquitetura do *framework* Metamorfose.

O componente *Conjunto de Entidades* mantém uma coleção de entidades que são manipuladas pelo *framework*. Essas entidades podem ser carregadas de fontes externas ou criadas através de transformações de dados executadas pelo Metamorfose. O componente *Funções do Framework* encapsula o conjunto de funções de manipulação de dados, incluindo funções para *carregar*, *filtrar*, executar operações *Map*, *MapReduce*, e *persistir*. Por meio dessas funções o usuário carrega entidades, filtra os dados, executa transformações e persiste os resultados. O componente *Definição de Mapeamentos* permite ao usuário especificar mapeamentos entre *esquema origem* e *esquema destino*, e definir transformações de dados por meio de FDUs. As FDUs são executadas através das funções *Map* e *MapReduce*, chamando métodos do *Apache Spark* (Zaharia et al., 2016) para executar as operações necessárias. A escolha do *Apache Spark* foi motivada pelo seu amplo suporte a diferentes cargas de processamento e abstrações para manipular conjuntos de dados. No entanto, o Metamorfose é independente, sendo possível utilizar outro componente para executar as transformações de dados.

A seguir serão apresentados detalhes sobre os componentes *Funções de Transformação MapReduce*, *Definição de Mapeamentos* e *Funções Definidas pelo Usuário (FDUs)*.

5.1.1.1 *Funções de Transformação MapReduce*

A unidade básica de transformação é a entidade. Uma entidade pode ser definida como $Et = \{I_1, I_2, ..., I_n\}$, em que I_i é uma instância de Et. Transformações de dados são executadas através de funções *Map* e *MapReduce*. Ambas funções recebem como parâmetros a entidade de origem e mapeamentos que definem as transformações de dados. Os mapeamentos serão apresentados na próxima seção. A função *Map* tem cardinalidade *1:1*, ou seja, ela retorna uma instância da entidade destino (I_{t_i}) para cada instância da entidade de origem (I_{s_i}). A função *MapReduce* tem cardinalidade *N:1*, ou seja, essa função retorna uma instância da entidade destino (I_{t_i}) para cada grupo de instâncias da entidade origem ($I_{s_1}, I_{s_2}, ..., I_{s_n}$). Para agrupar as instâncias da entidade origem é necessário fornecer uma chave de agrupamento como parâmetro. Transformações com cardinalidade *N:N* não são suportadas diretamente pelo *framework*, mas é possível definir uma combinação de funções *Map* e *MapReduce* para possibilitar esse tipo de transformação.

A Figura 5.3 ilustra o fluxo de execução da função *Map*. A instância de origem I_{s_1} é transformada na instância de destino I_{t_1}. O conjunto de mapeamentos $(M_1, M_2, ..., M_n)$ define como os campos de I_{s_1} serão transformados em campos da entidade I_{t_1}.

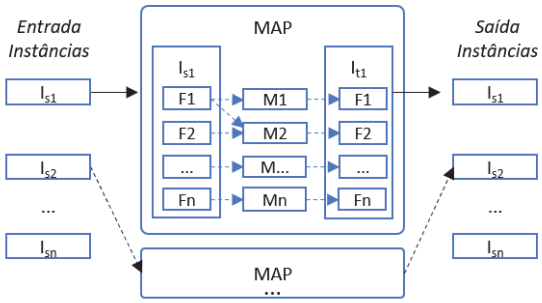


Figura 5.3: Fluxo de execução da função *Map*.

A Figura 5.4 ilustra o fluxo de execução da função *MapReduce*. As instâncias de origem I_{s_1} e I_{s_2} serão transformadas em pares de chave-valor pela função *map* e reduzidas para a instância destino I_{t_1} pela função *reduce*. Na fase de redução, os mapeamentos $(M_1, M_2, ..., M_n)$ são aplicados para cada instância do grupo.

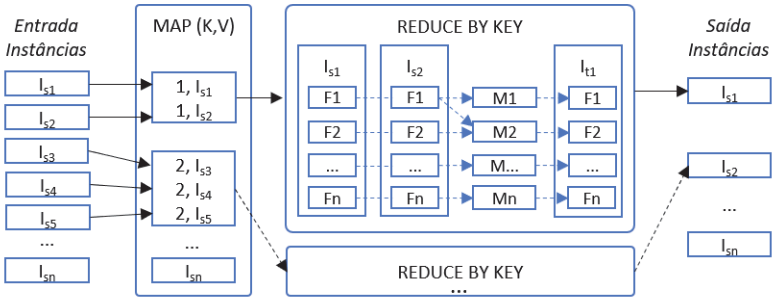


Figura 5.4: Fluxo de execução da função *MapReduce*.

5.1.1.2 *Definição de Mapeamentos*

Mapeamentos são declarações que estabelecem correspondência entre campos de origem e destino, que podem ser integrados com funções de transformação de dados. Uma entidade

tem um esquema de dados que pode ser definido como $S = s_1, ..., s_n$, em que s_i representa um campo de dados do esquema S. A partir de um esquema S é possível derivar um esquema destino T aplicando transformações sobre os campos de S. Transformações podem ser representadas como um conjunto de mapeamentos $M = \{(s_1, t_1, f_1), ..., (s_n, t_n, f_n)\}$, em que s_i representa um ou mais campos de origem $\{s_{i_1}, ..., s_{i_j}\}$, t_i um ou mais campos de destino $\{t_{i_1}, ..., t_{i_j}\}$ e f_i uma função definida pelo usuário (FDU) para transformar os dados. Por exemplo, considerando uma entidade $people(id, fname, lname)$ e o mapeamento $M_1 = (s(id, fname, lname), t(id, name), f(jscript))$, a função f recebe como parâmetros os campos *id*, *fname* e *lname* e deve retornar os campos *id* e *name*, conforme lógica de transformação implementada em f. A variável $jscript$ representa o código-fonte da FDU.

Os mapeamentos são usados em funções *Map* e *MapReduce* para transformar instâncias de origem em instâncias de destino. O Metamorfose fornece três diferentes tipos de FDUs para traduzir campos de entrada: *Casting*, *Java* ou *Javascript*. *Casting* representa conversões de tipos de dados (*string* para inteiro e inteiro para *string*, por exemplo). Transformações complexas devem ser implementadas usando FDUs em Java ou Javascript. Transformações do tipo *Casting* permitem apenas mapeamentos um-para-um entre campos de origem e destino. Transformações usando FDUs permitem mapeamentos um-para-um, um-para-muitos, muitos-para-um e muitos-para-muitos. Internamente, o *framework* usa objetos JSON para representar as instâncias das entidades de origem e destino. As FDUs devem receber um objeto JSON e retornar um objeto JSON.

5.1.1.3 Funções Definidas pelo Usuário (FDUs)

As funções definidas pelo usuário em um mapeamento (*Definição de Mapeamentos*) podem ser implementadas em Java ou Javascript. As FDUs definidas em Java devem ser compiladas e adicionadas ao *classpath* do Metamorfose. O usuário deve criar uma classe Java e implementar uma interface fornecida pelo *framework*, contendo a lógica de transformação. No momento de definir um mapeamento, o usuário deve informar o nome da classe criada para que o *framework* possa invocar a FDU. Em contrapartida, a implementação de FDU em Javascript é adicionada diretamente na definição do mapeamento como texto. Ambos os tipos de FDUs recebem por parâmetro os campos e valores da instância que será transformada, em formato JSON. O valor de retorno após a execução das operações de transformação de dados deve ser um objeto JSON contendo os campos definidos no mapeamento.

A Figura 5.5 apresenta dois exemplos de FDUs, uma em Javascript e outra em Java. Ambas FDUs recebem um objeto de entrada (ou instância), executam a lógica de transformação e retornam um objeto de saída (ou instância). A execução dessas FDUs é realizada por meio de uma função *Map*. Na Figura 5.5 (a), a variável $values$ é convertida no objeto JSON $input$. Esse objeto é usado para acessar os campos e valores da instância que será transformada. O resultado da transformação de dados deve ser armazenado no objeto JSON $output$. O usuário pode utilizar os recursos da linguagem Javascript para expressar a lógica de transformação. O objeto $output$ pode ser composto por atributos atômicos, por objetos JSON e por *arrays* de objetos JSON. De forma análoga, a Figura 5.5 (b) apresenta a mesma FDU implementada em linguagem Java.

```javascript
function f1(values) {
    input = JSON.parse(values);
    output = JSON.parse('{}');
    // Begin: User Transformation Logic
    output.field1 = input.field1;
    output.field2 = input.field2;
    output.field3 = 0;
    if (input.field3 > 0)
        output.field3 = input.field3;
    ...
    // End: User Transformation Logic
    return JSON.stringify(output);
}
```

(a) FDU em Javascript (*Map*).

```java
public Object executeUDF(Object values) {
    JSONObject input = new JSONObject(values.toString());
    JSONObject output = new JSONObject();
    // Begin: User Transformation Logic
    output.put("field1", input.get("field1"));
    output.put("field1", input.get("field1"));
    output.put("field1", 0);
    if (input.getInt("field3") > 0)
        output.put("field1", input.get("field1"));
    // ...
    // End: User Transformation Logic
    return output.toString();
}
```

(b) FDU em Java (*Map*).

Figura 5.5: Funções definidas pelo usuário em Javascript e Java, para execução por meio de função *Map*.

Na Figura 5.6 são exibidas FDUs em Javascript e Java para transformações de dados executadas por meio da função *MapReduce*. Nesse caso, a FDU é executada na fase de redução (função *reduce*) e esse aspecto deve ser considerado ao implementar a lógica de transformação. De forma semelhante ao exemplo da Figura 5.5, a variável *values* é convertida no objeto JSON *input*. No entanto, *input* contém dois objetos encapsulados (*object*1 e *object*2) que devem ser reduzidos para apenas um objeto *output*. Como a função *reduce* é executada para todas as instâncias do agrupamento, o objeto *output* é utilizado como entrada para as próximas chamadas da função.

```javascript
function f2(values) {
    input = JSON.parse(values);
    output = JSON.parse('{}');
    // Begin: User Transformation Logic
    output.field1 = input.object1.field1
                  + input.object2.field1;
    output.field2 = input.object1.field2
                  + input.object2.field2;
    output.field3 = "value";
    ...
    // End: User Transformation Logic
    return JSON.stringify(output);
}
```

(a) FDU em Javascript (*MapReduce*).

```java
public Object executeUDF(Object values) {
    JSONObject input = new JSONObject(values.toString());
    JSONObject output = new JSONObject();
    // Begin: User Transformation Logic
    int result1 = input.getJSONObject("object1").getInt("field1")
            + input.getJSONObject("object2").getInt("field1");
    output.put("field1", result1);
    int result2 = input.getJSONObject("object1").getInt("field2")
            * input.getJSONObject("object2").getInt("field2");
    output.put("field2", result2);   // ...
    // End: User Transformation Logic
    return output.toString();
}
```

(b) FDU em Java (*MapReduce*).

Figura 5.6: Funções definidas pelo usuário em Javascript e Java, para execução por meio de função *MapReduce*.

Ambos os tipos de FDUs (Javascript e Java) operam de maneira semelhante, no entanto, o uso de Javascript fornece flexibilidade ao usuário. Não há necessidade de recompilar os mapeamentos toda vez que novas implementações são fornecidas. Em adição, como os mapeamentos podem ser persistidos como arquivos JSON, o uso de Javascript facilita a sua reutilização em outros cenários.

5.1.2 Fluxo de Execução do Metamorfose

O Metamorfose tem um fluxo de execução composto por 10 etapas, apresentado na Figura 5.7. Em (1) os dados são carregados como um *dataset Spark*. Baseado nos mapeamentos fornecidos pelo usuário o *dataset* é transformado (2) produzindo o (3) *dataset'* como resultado. O *dataset'* pode ser usado: (4) como entrada para uma nova transformação, (5) como entrada para operação de filtro (consulta SQL) ou (6) pode ser persistido como um arquivo CSV, JSON ou tabela de banco de dados relacional. A execução de consulta SQL sobre o *dataset'* produz

um novo (7) *dataset"*. O *dataset"* pode ser usado para novas transformações de dados (8), para persistência (9) ou para visualização (10).

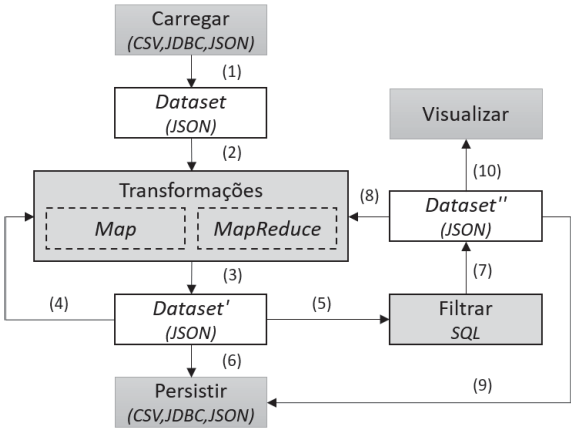


Figura 5.7: Fluxo de execução do *framework* Metamorfose.

Esse fluxo permite definir cadeias de transformações de dados de forma flexível. As transformações podem ser aplicadas sobre o *dataset* corrente ou podem criar um *dataset* novo com os dados resultantes.

5.1.3 Exemplos de Transformação

Esta seção apresenta dois exemplos de transformação de dados por meio do *framework* Metamorfose. Os exemplos mostram a definição de mapeamentos e uso das funções *Map* e *MapReduce* sobre um conjunto de dados de pessoas (*people*). Esse conjunto de dados é composto por seis instâncias, em que cada instância é definida pelos campos *fname*, *lname*, *child* e *sex*.

A Figura 5.8 apresenta um exemplo de transformação usando a função *Map*. Como entrada são fornecidos o conjunto de instâncias *people* e o conjunto de mapeamentos *maps*. A tabela *Definição de Mapeamentos* apresenta os mapeamentos entre campos de origem e destino, o tipo de transformação (JavaUDF, JsUDF ou Casting) e FDUs utilizadas para transformação de dados. Os campos *fname* e *lname* são mapeados para o campo *name*, por meio de uma FDU em Java. O caminho *"metamorfose.ConcatStrings"* aponta para uma classe Java que concatena os campos *fname* e *lname* para produzir o campo *name*. O campo *child* é mapeado diretamente para a entidade destino (*Casting*). O campo *sex* é transformado para valores numéricos e mapeado para o campo *sex* na entidade destino usando a FDU sex. Para executar as transformações de dados através do *framework*, primeiramente o conjunto de dados (*people*) é carregado em memória e depois a função *Map* é invocada recebendo por parâmetro os mapeamentos definidos pelo usuário e instâncias de dados. Os resultados da transformação são observados na parte inferior da Figura 5.8, como um conjunto de documentos JSON.

A Figura 5.9 apresenta um exemplo de transformação usando a função *MapReduce*. A abordagem de transformação é semelhante a apresentada no exemplo anterior (*Map*). No entanto, a função *MapReduce* agrupa as instâncias da entidade origem de acordo com chave de agrupamento fornecida pelo usuário e reduz cada grupo para uma instância da entidade destino. No exemplo da Figura 5.9, as instâncias de *people* são agrupadas pela chave *fname*. Na tabela *Definição de Mapeamentos* os campos *fname*, *lname* e *sex* são mapeados diretamente para a entidade destino (*Casting*). O campo *child* é mapeamento para o campo *children* por meio da FDU em Javascript $childrenArray$. Essa FDU é usada para criar um *array* chamado *children*, de tal forma que o valor do campo *child* de cada instância de *people*, pertencente a

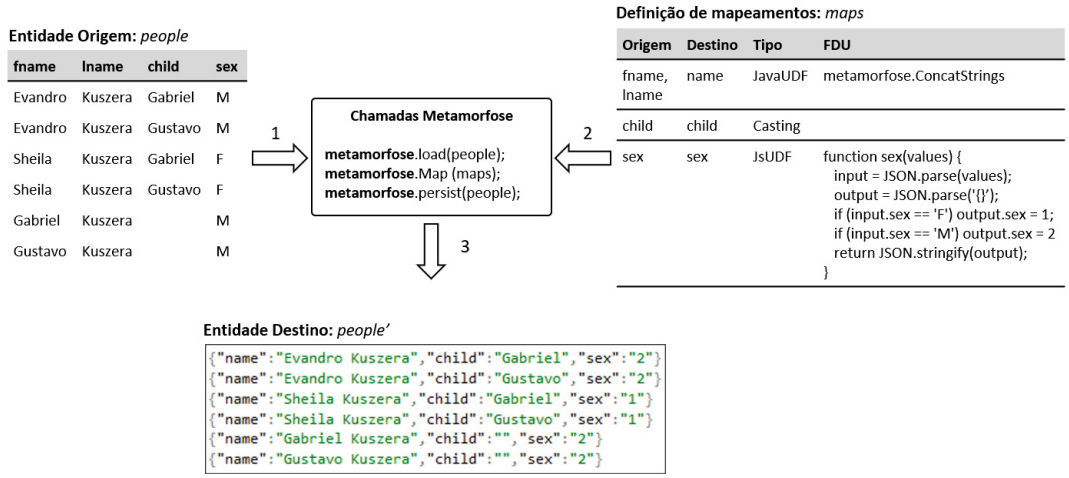


Figura 5.8: Exemplo de transformação de dados do Metamorfose por meio da função *Map*.

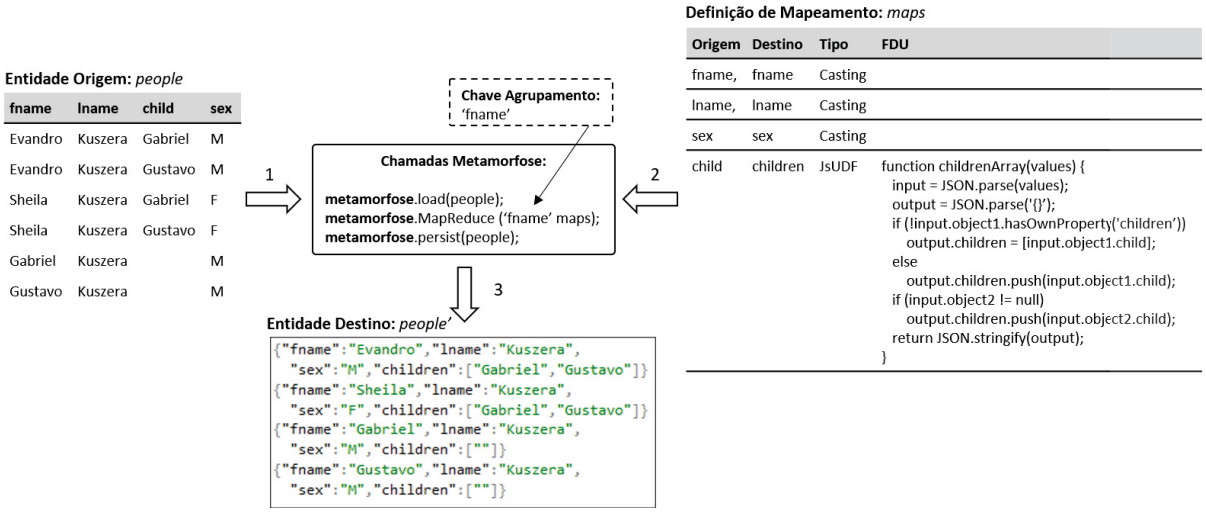


Figura 5.9: Exemplo de transformação de dados do Metamorfose por meio da função *MapReduce*.

mesma chave de agrupamento, é inserido como um elemento do *array children*. Os resultados da transformação são observados na Figura 5.9 como um conjunto de documentos JSON. Esse exemplo ilustra como usar a função *MapReduce* para implementar transformações que produzem dados aninhados.

As duas próximas seções apresentam o mecanismo de tradução de DAGs para comandos do Metamorfose.

5.2 *GERADOR DE COMANDOS*

O componente *Gerador de Comandos* recebe um DAG e gera um conjunto de comandos de transformação que será usado pelo *framework* Metamorfose para transformar instâncias RDB em instâncias NoSQL. A Figura 5.10 exibe exemplos de DAGs suportados pelo processo de conversão.

Cada DAG representa uma entidade NoSQL, em que o vértice raiz define o nome desta entidade. O *DAG*1 é composto de dois vértices, *Tabela A* e *Tabela B*. Nesse caso, as instâncias da *Tabela B* (lado muitos) serão aninhadas nas respectivas instâncias da *Tabela A* para gerar a nova instância NoSQL. No *DAG*2, as instâncias da *Tabela A* (lado um) serão aninhadas nas

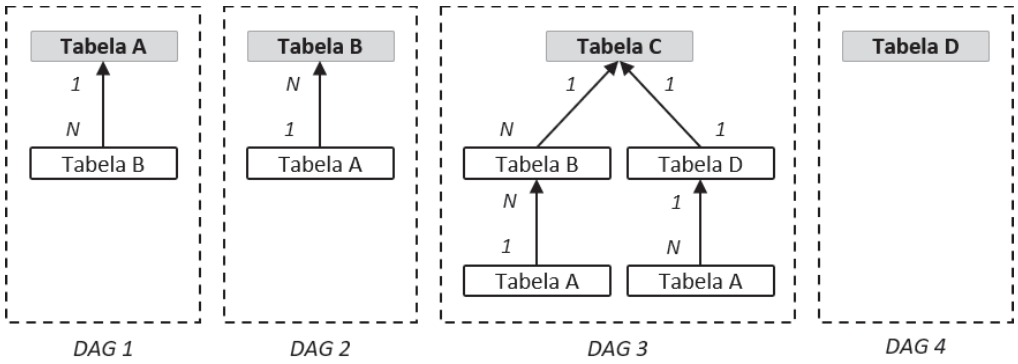


Figura 5.10: Exemplos de DAGs suportados pelo componente Gerador de Comandos.

respectivas instâncias da *Tabela B*. No $DAG3$ há dois vértices representando a *Tabela A*. Essa situação é permitida para possibilitar a geração de esquemas redundantes. As instâncias da *Tabela A* serão aninhadas na *Tabela B*, e depois na *Tabela D*. Depois, as instâncias resultantes da *Tabela B* e da *Tabela D* serão aninhadas na *Tabela C*. O $DAG4$ tem apenas um vértice e, neste caso, as instâncias da *Tabela D* serão convertidas diretamente para o formato do modelo NoSQL alvo. O formato do aninhamento é dependente da direção do relacionamento (*1-1, 1-N, N-1*) e do modelo NoSQL destino, sendo que para cada modelo são usadas diferentes estratégias de conversão de instâncias. Relacionamentos *M:N* não são suportados diretamente no DAG, mas podem ser representados como dois relacionamentos *1:N*.

Cada comando de transformação gerado pelo componente *Gerador de Comandos* encapsula operações para aninhar tabelas (vértices no DAG), de tal forma que um vértice folha seja aninhado em seu vértice sucessor. Esse processo se repete até que todos os vértices sejam aninhados no vértice raiz. Um comando de transformação pode ser definido como $command = (joinSpec, fieldMaps, function)$, em que $joinSpec$ representa a operação de junção entre tabelas que precisam ser desnormalizadas, $fieldMaps$ o conjunto de mapeamentos entre campos das tabelas, e $function$ o tipo da função de transformação que será invocada. O Algoritmo 1 apresenta os passos para percorrer o DAG e gerar os comandos de transformação.

O Algoritmo 1 percorre o conjunto de vértices do DAG como uma árvore, no sentido vértice folha $\rightarrow$ vértice raiz. Para cada vértice folha um comando é criado, encapsulando operações para aninhar o vértice folha em seu respectivo vértice sucessor (ou vértice pai) no DAG. Após a criação do comando o vértice folha é removido do DAG. O algoritmo encerra quando o DAG é reduzido a um vértice (vértice raiz), que representa a entidade NoSQL alvo. A forma como os dados serão aninhados depende do modelo NoSQL destino. Neste trabalho foram implementadas duas estratégias para transformar dados relacionais em dados aninhados. Para cada estratégia foi implementado um algoritmo específico, chamado a partir do Algoritmo 1, linhas 9 e 11, respectivamente. A primeira estratégia transforma os dados relacionais para documentos JSON, de acordo com o modelo NoSQL orientado a documentos. A segunda transforma os dados em documentos JSON de acordo com o modelo NoSQL orientado a famílias de colunas. Essas estratégias serão apresentadas a seguir. Ademais, novas estratégias podem ser implementadas, permitindo estender o *framework* para novos cenários.

5.2.1 Conversão para NoSQL Orientado a Documentos

O Algoritmo 2 define a estratégia para conversão de dados relacionais para NoSQL orientado a documentos. As tabelas RDB são transformadas em objetos JSON e os relacionamentos

Algoritmo 1 Construção de comandos (*commands*) de transformação:

Input: DAG e modelo NoSQL destino (documento ou família de colunas).
Output: Conjunto de comandos de transformação.

```
1: commands ← empty
2: if graph.vertexSet = 1 then
3:     simple_cmd(graph.getVertex(0))
4: else
5:     while graph.vertexSet > 1 do
6:         leafVertexs ← graph.getVertexsWithInDegree(0)
7:         for all leaf ∈ leafVertexs do
8:             if targetNoSQL = Document then
9:                 commands.add(nosql_doc_cmd(leaf))
10:            else if targetNoSQL = ColumnFamily then
11:                commands.add(nosql_col_cmd(leaf, graph.vertexSet))
12:            end if
13:            graph.remove(leaf)
14:        end for
15:    end while
16: end if
17: return commands
```

entre tabelas são representados por referências (similar ao modelo relacional), objetos JSON embutidos ou *arrays* de objetos JSON embutidos.

Algoritmo 2 NoSQL Doc Command:

Input: Vértice Folha (*leaf*).
Output: Comando de Transformação (*Command*).

```
1: fieldMaps ← empty
2: succr ← leaf.getSuccessor()
3: for all field ∈ succr.fields do
4:     fieldMaps.oneToOne(field, field, casting)
5: end for
6: if leaf.isOneSide then
7:     jsUDF ← JScriptGenerator(leaf, docEmbedded)
8:     function ← "map"
9: else if leaf.isManySide then
10:    jsUDF ← JScriptGenerator(leaf, arrayEmbedded)
11:    function ← "mapreduce"
12: end if
13: fieldMaps.manyToOne(leaf.fields, leaf.name, jsUDF)
14: succr.addField(leaf.name)
15: joinSpecAdd(leaf, succr)
16: return Command(joinSpec, fieldMaps, function)
```

O Algoritmo 2 recebe o vértice folha como parâmetro e retorna um comando que encapsula operações para aninhar a tabela vértice folha na tabela vértice sucessor. O formato do aninhamento depende do lado do relacionamento em que a tabela representada pelo vértice folha se encontra (linhas 6 – 12). Se o vértice folha está no *lado um*, as instâncias serão aninhadas como documentos JSON embutidos. Se está no *lado muitos*, então as instâncias serão aninhadas como *arrays* de documentos JSON. Este processo é repetido até que o vértice raiz contenha todos

os demais vértices do DAG (laço *while* do Algoritmo 1). A função *JScriptGenerator* gera automaticamente FDUs em Javascript a partir dos metadados do vértice e da aresta. As FDUs geradas contêm comandos para transformar as instâncias das tabelas RDB em objetos JSON. A função *JScriptGenerator* pode ser estendida para incluir novas estratégias de conversão e modificar o formato do arquivo JSON gerado.

5.2.2 Conversão para NoSQL Orientado a Família de Colunas

O Algoritmo 3 define a estratégia para conversão de dados relacionais para NoSQL orientado a família de colunas. As tabelas RDB são desnormalizadas e transformadas em tabelas compostas por famílias de colunas. Para representar a estrutura e dados da tabela NoSQL são usados objetos JSON, em que cada objeto JSON é composto por um campo *rowkey* (identificador da instância) e um conjunto de documentos JSON embutidos para representar as famílias de colunas.

Algoritmo 3 NoSQL Col Command:

Input: Vértice folha (*leaf*) e número de vértices do grafo (*verticesNumber*).
Output: Comando de transformação (*Command*).

```
 1: fieldMaps ← empty
 2: root ← getRootVertex()
 3: joinSpec ← createJoinSpec(leaf, root)
 4: if verticesNumber = 2 then
 5:     fieldMaps.oneToOne(root.PK, "rowkey", casting)
 6:     jsUDF ← JScriptGenerator(root, docEmbedded)
 7:     fieldMaps.manyToOne(root.fields, root.name, jsUDF)
 8: end if
 9: for all field ∈ root.fields do
10:     fieldMaps.oneToOne(field, field, casting)
11: end for
12: nesting_key = leaf.PK
13: jsUDF ← JScriptGenerator(leaf, nesting, nesting_key)
14: fieldMaps.manyToOne(leaf.fields, leaf.name, jsUDF)
15: root.addField(leaf.name)
16: return Command(joinSpec, fieldMaps, "mapreduce")
```

O Algoritmo 3 recebe o vértice folha, o número de vértices do DAG e retorna um comando encapsulando as operações para aninhar a tabela vértice folha diretamente no vértice raiz, como uma família de colunas. Primeiro é definida a operação de junção entre o vértice folha e vértice raiz, incluindo os vértices intermediários (linha 3). Nos comandos seguintes (linhas 9-15) a tabela vértice folha é aninhada como uma família de colunas na tabela vértice raiz. A variável *nesting_key* é usada para renomear os campos do vértice folha, permitindo aninhar as instâncias do lado muitos do relacionamento dentro de uma família de colunas sem duplicar o nome dos campos. Quando o número de vértices do DAG é igual a dois (linhas 4 – 7), significa que este será o último comando criado e algumas operações extras devem ser executadas. A primeira é a criação do campo *row_key* da instância NoSQL. A chave primária (PK) da tabela vértice raiz é mapeada como campo *row_key*. A segunda é a criação da família de colunas da tabela vértice raiz. A função *JScriptGenerator* gera automaticamente FDUs em Javascript a partir dos metadados do vértice e aresta. As FDUs contêm comandos para transformar as tabelas do RDB como famílias de colunas na tabela NoSQL. Na linha 14, a

função *JScriptGenerator* recebe a variável *nesting_key* para renomear os campos das famílias de colunas e evitar a ocorrência de nomes duplicados.

5.3 *EXECUTOR DE COMANDOS*

Este componente recebe um conjunto de comandos gerados na etapa anterior, pelo componente *Gerador de Comandos*. Esse conjunto de comandos é processado e convertido em chamadas *Load*, *Map*, *MapReduce* e *Persist* do *framework* Metamorfose. Além disso, o componente *Executor de Comandos* encapsula parâmetros de conexão com o banco de dados relacional e banco de dados NoSQL, e controla a ordem de execução dos comandos de transformação de dados.

5.4 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Neste capítulo foi apresentado um processo para converter dados relacionais em dados semiestruturados, considerando os modelos de dados NoSQL orientados a documentos e famílias de colunas. O processo usa grafos acíclicos direcionados (DAG) como abstração para representar o modelo de dados de destino. O DAG é utilizado para definir quais tabelas RDB serão transformadas e migradas como uma entidade NoSQL. O *framework* Metamorfose recebe como entrada um conjunto de DAGs e executa as transformações de dados através de funções *Map* e *MapReduce*, produzindo objetos JSON para representar as instâncias em formato semiestruturado. O *framework* tem pontos de extensão, permitindo adicionar novas estratégias de conversão de dados, através da personalização do mecanismo de geração de funções definidas pelo usuário (FDU) em Javascript. Ademais, os mapeamentos e FDUs gerados podem ser persistidos para fins de auditoria das transformações de dados ou para uso futuro.

No próximo capítulo serão apresentados experimentos para validar o *framework* Metamorfose e o processo de conversão de RDB para NoSQL.

6 EXPERIMENTOS: USO DO METAMORFOSE PARA CONVERSÃO DE DADOS

Neste capítulo são apresentados dois experimentos para validar o processo de conversão e migração de dados definido nesta tese[1]. O objetivo é validar o Metamorfose no processo de conversão de RDB para NoSQL. O primeiro experimento foca na transformação de dados relacionais. O segundo experimento tem foco na transformação de uma base relacional para bases NoSQL orientadas a documentos e famílias de colunas.

6.1 EXPERIMENTO I: CONVERTENDO DADOS ESTRUTURADOS

O *framework* Metamorfose foi utilizado em um cenário de transformação de dados abertos fornecidos pelo Instituto Nacional de Educação e Pesquisa Anísio Teixeira (INEP) [2]. Esses dados fornecem informações sobre a educação brasileira, incluindo número de matrículas, professores, escolas e cursos em arquivos CSV que ultrapassam 69 milhões de registros por ano. No entanto, variações no formato dos dados são encontradas a cada versão disponibilizada pelo governo, como adição de novos campos, alteração e remoção de campos existentes. Nesse contexto, o *framework* foi utilizado para mapear e transformar os dados em formato CSV para um esquema relacional pré-definido. O *framework* e resultados obtidos foram publicados na trilha demo do SBBD'18 (Kuszera et al., 2018).

Para facilitar a interação com o usuário foi desenvolvida uma interface gráfica para auxiliar na definição dos mapeamentos e funções de transformação. A Figura 6.1 apresenta a tela principal do *framework*.

Em (1) o usuário pode carregar e remover *datasets*, visualizar a lista de *datasets* disponíveis, ou adicionar mapeamentos para os *datasets* existentes. O botão *Add Mapping* exibe a tela de mapeamento (detalhes a seguir). O usuário carrega *datasets* a partir de arquivos CSV ou tabelas de bancos relacionais (por exemplo, Postgres). Em (2) o usuário pode submeter consultas SQL (usando módulo *Spark SQL*) sobre os *datasets* carregados. Em (3) são exibidos os dados produzidos pela consulta SQL. O resultado da consulta SQL pode ser adicionado como um novo *dataset* na lista de *datasets* disponíveis (botão *To Dataset*). No exemplo da Figura 6.1 é exibido uma consulta SQL sobre os dados de um arquivo CSV (*matriculas2013*). Os resultados são exibidos na tabela da parte inferior da tela.

Para definir mapeamentos entre entidades de origem e destino é preciso selecionar um *dataset* da lista e clicar no botão *Add Mapping*. Essa ação exibe a tela de mapeamentos e carrega o esquema do *dataset* selecionado. A Figura 6.2 apresenta um exemplo de mapeamento.

Em (1) são exibidos os campos e tipos de dados do *dataset* de origem. Em (2) o usuário especifica os campos e tipos de dados do *dataset* de destino. Em (3) define o tipo de transformação (*Casting*, *Java* or *Javascript*). Para transformações usando FDUs em Javascript o usuário deve

[1]Parte do conteúdo deste capítulo foi publicado em:

- Kuszera, E. M., Peres, L. M. e Fabro, M. D. D. (2018). Metamorfose: a Data Transformation Framework Based on Apache Spark. Em 33rd Annual Brazilian Symposium on Databases: Proceeedings Companion, SBBD 2018 Companion, Rio de Janeiro, RJ, Brazil, August 25-26, 2018., páginas 11–16.
- Kuszera, E. M., Peres, L. M. e Fabro, M. D. D. (2019). Toward RDB to NoSQL: Transforming Data with Metamorfose Framework. Em Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19, página 456–463, New York, NY, USA. Association for Computing Machinery.

[2]INEP: `http://www.inep.gov.br/`

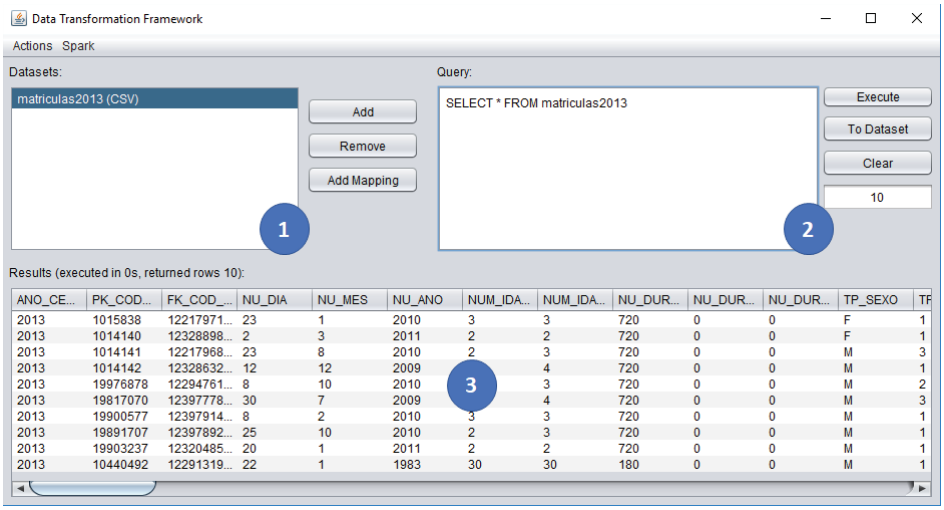


Figura 6.1: Tela principal do *framework* Metamorfose.

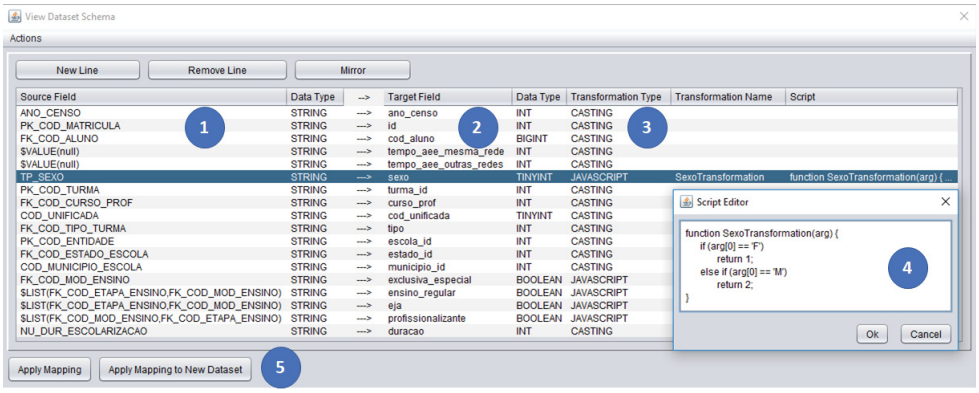


Figura 6.2: Tela de definição de mapeamentos entre esquemas de origem e destino.

inserir o código na tela de edição de *script* (4). Após a definição de mapeamentos, duas opções estão disponíveis (5): através do botão *Apply Mapping* os mapeamentos são aplicados sobre o *dataset* origem e modificam seu esquema de dados, através do botão *Apply Mapping to New Dataset* um novo *dataset* é criado a partir do *dataset* de origem. Esse fluxo de trabalho permite definir cadeias de transformação de dados, em que o resultado de uma transformação pode ser usado como entrada para a próxima. As transformações de dados são executadas somente quando o usuário submete uma consulta SQL ou quando persiste os dados do *dataset* destino. Essas duas ações disparam chamadas para funções do *Apache Spark*.

Para validar o protótipo implementado foram realizados experimentos usando dados de matrículas realizadas em todo o Brasil do ano de 2013. Os dados foram disponibilizados em cinco arquivos CSV representando cada região do país: Sul, Sudeste, Centro-Oeste, Norte e Nordeste. Foram definidos 82 mapeamentos entre campos do CSV e campos do esquema relacional destino. Mapeamentos um-para-um, muitos-para-um e FDUs em Javascript foram usadas para transformar os dados. O protótipo foi executado em uma máquina com processador Intel Core i7 2.5GHz, 16 GB de RAM, Windows 10 Home e banco de dados Postgres 10. A Tabela 6.1 mostra o número de registros por região do país e tempo de execução para ler, transformar e inserir os dados no Postgres.

O objetivo deste experimento foi verificar a viabilidade da implementação do *framework* sobre dados de tamanho moderado. É possível observar que o tempo de execução cresce de forma linear conforme número de registros.

Tabela 6.1: Número de registros e tempo de execução para transformar os dados.

Região do Brasil	Registros	Tempo
Centro-Oeste	4.038.979	10 min
Sul	7.276.108	18 min
Nordeste	16.729.543	41 min
Sudeste + Norte	27.379.690	65 min

6.2 EXPERIMENTO II: CONVERTENDO RDB PARA NOSQL

Neste experimento o *framework* Metamorfose foi usado para converter dados relacionais para os modelos NoSQL orientados a documentos e famílias de colunas. O objetivo principal é validar a etapa **4. Migração de Dados** da abordagem de conversão, em que DAGs são usados para definir o esquema NoSQL destino e gerar os comandos de transformação de dados do Metamorfose.

6.2.1 Cenário do Experimento

O banco de dados relacional usado no experimento é proveniente na aplicação *Dell DVD Store* (Jaffe e Muirhead, 2005), usada para avaliar o desempenho de servidores *Dell*. A aplicação simula um comércio eletrônico de DVDs e fornece um conjunto de *scripts* configuráveis para popular as tabelas com registros fictícios sobre vendas de DVDs. A Figura 6.3 mostra o diagrama entidade-relacionamento do banco de dados da aplicação, composto por sete entidades.

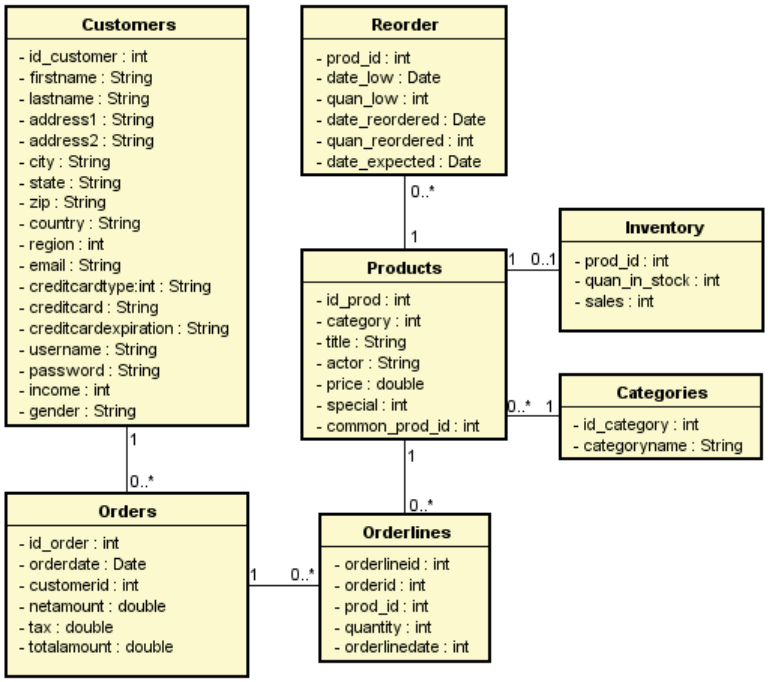


Figura 6.3: Diagrama Entidade-Relacionamento do banco de dados da aplicação *Dell DVD store*.

Para demonstrar a abordagem de conversão serão consideradas apenas as entidades *Orders*, *Orderlines* e *Products*. Três bancos de dados foram configurados no Postgres 10, nomeados como *RDB1*, *RDB2* e *RDB3*, e populados com números de registros distintos. A lista dos bancos de dados e número de registros por tabela são apresentados na Tabela 6.2.

Tabela 6.2: Bancos de dados gerados e respectivos números de registros por tabela.

RDB Tam.	Orders	Orderlines	Products
RDB1	12.000	60.350	10.000
RDB2	60.000	301.188	50.000
RDB3	120.000	601.009	100.000

A partir dessas três tabelas foram construídos os DAGs *Orders*, *Orderlines* e *Products* apresentados na Figura 6.4.

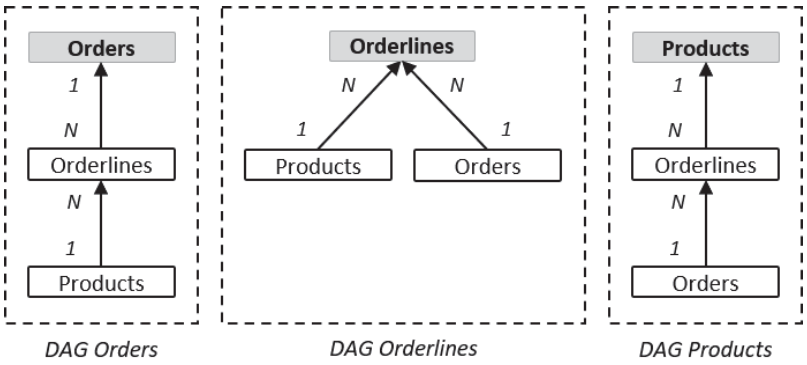


Figura 6.4: DAGs *Orders*, *Orderlines* e *Products* usados como entrada no processo de conversão e migração de dados.

O vértice raiz do DAG representa a entidade NoSQL alvo e os demais vértices representam as entidades que serão aninhadas. Por exemplo, no DAG *Orders*, as instâncias de *Products* e *Orderlines* serão aninhadas em *Orders*. O formato do aninhamento depende do modelo NoSQL selecionado. Todos os três DAGs preservam os dados de entrada nos dados de saída. Porém, é possível criar DAGs representando somente partes da informação, dependendo do domínio da aplicação.

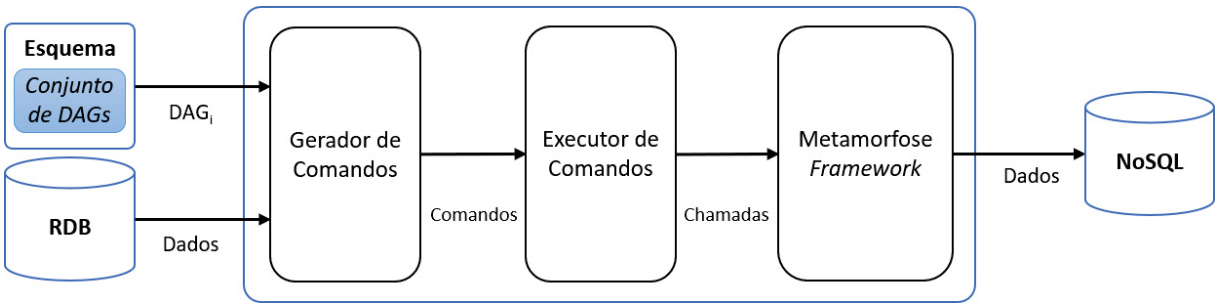


Figura 6.5: Processo de conversão e migração de dados de RDB para NoSQL, apresentado no Capítulo 5.

A Figura 6.5 reexibe o processo de conversão e migração de dados apresentado no Capítulo 5 para facilitar a leitura. Esse processo é usado para converter as instâncias do RDB para NoSQL, recebendo como entrada os *i)* parâmetros de conexão do RDB, o *ii)* conjunto de DAGs e o *iii)* modelo de dados NoSQL destino. Primeiro, o componente *Gerador de Comandos* converte os DAGs em comandos de transformação de dados. Na sequência, o componente *Executor de Comandos* converte os comandos em chamadas às funções do Metamorfose, para ler os dados do RDB, transformar em objetos JSON e persistir em formato NoSQL. A execução

dos experimentos foi realizada em uma máquina com Windows 10 Professional, Intel Core i7 2.5GHz, 16 GB de RAM e 1 TB de disco.

6.2.2 Resultados dos Experimentos

Nesta seção serão apresentados os artefatos e resultados produzidos para converter o DAG *Orders* para NoSQL orientado a documentos e NoSQL orientado a família de colunas. A conversão dos demais DAGs (*Orderlines* e *Products*) não será exibida pelo fato do processo ser semelhante. No final desta seção serão apresentados resultados preliminares de desempenho da abordagem para converter todos os DAGs para ambos os modelos NoSQL.

6.2.2.1 RDB para NoSQL Orientado a Documentos

Abaixo são apresentados os mapeamentos e comandos de transformação de dados gerados para converter instâncias do RDB de acordo com o DAG *Orders* da Figura 6.4. Lembrando que mapeamentos são declarações que estabelecem correspondência entre campos de origem e destino, que podem ser integrados com funções de transformação de dados, e um comando de transformação é definido como $command = (joinSpec, fieldMaps, function)$, em que $joinSpec$ representa a operação de junção entre tabelas que precisam ser desnormalizadas, $fieldMaps$ o conjunto de mapeamentos entre campos das tabelas, e $function$ o tipo da função de transformação que será invocada (*Map* ou *MapReduce*).

$m_{1.1} = (s_{(orderlineid)}, t_{(orderlineid)}, f_{casting})$;
$m_{1.2} = (s_{(orderid)}, t_{(orderid)}, f_{casting})$;
$m_{1.3} = (s_{(prod\_id)}, t_{(prod\_id)}, f_{casting})$;
$m_{1.4} = (s_{(quantity)}, t_{(quantity)}, f_{casting})$;
$m_{1.5} = (s_{(id\_prod,category,title,actor,price,special)}, t_{(products)}, f_{docEmbedded})$;
$cmd_1 = ((Products \bowtie Orderlines), fieldMaps, Map)$

$m_{2.1} = (s_{(id\_order)}, t_{(id\_order)}, f_{casting})$;
$m_{2.2} = (s_{(customerid)}, t_{(customerid)}, f_{casting})$;
$m_{2.3} = (s_{(orderdate)}, t_{(orderdate)}, f_{casting})$;
$m_{2.4} = (s_{(orderlineid,orderid,prod\_id,quantity,products)}, t_{(orderlines)}, f_{arrayEmbedded})$;
$cmd_2 = ((Orderlines \bowtie Orders), fieldMaps, MapReduce)$

Dois comandos são gerados, cmd_1 e cmd_2. O comando cmd_1 encapsula operações para transformar as instâncias de *Products* e *Orderlines* em objetos JSON e aninhar *Products* como um objeto embutido em *Orderlines*. O comando cmd_2 primeiro transforma a entidade *Orders* em objeto JSON e, então aninha *Orderlines* como um *array* de objetos JSON em *Orders*. O processo de criação de objetos JSON é encapsulado nas FDUs *docEmbedded* e *arrayEmbedded*.

A Figura 6.6 mostra o código-fonte das FDUs. Essas FDUs são geradas automaticamente pela função $JScriptGenerator$ (Algoritmo 2) a partir dos vértices e arestas do DAG. O usuário pode modificar o código gerado para aplicar customizações antes de executar as transformações.

Processo de execução dos comandos de transformação de dados produz instâncias de *Orders* como objetos JSON. A Figura 6.7 exibe uma instância de *Orders*, composta por um *array* de objetos *Orderlines*, em que cada instância de *Orderlines* tem uma instância de *Products* associada. As instâncias produzidas pelo Metamorfose podem ser persistidas em arquivos JSON ou diretamente no banco NoSQL, como uma coleção de documentos.

Oirgem	Destino	FDU

id_prod,
category,
title,
actor,
price,
special
	products	function docEmbedded(values) {
		input = JSON.parse(values);
		output = JSON.parse('{}');
		output.products = JSON.parse('{}');
		output.products.id_prod = input.id_prod;
		output.products.category = input.category;
		output.products.title = input.title;
		output.products.actor = input.actor;
		output.products.price = input.price;
		output.products.special = input.special;
		return JSON.stringify(output);
		}

Origem	Destino	FDU

orderlineid,
orderid,
prod_id,
quantity,
products
	orderlines	function arrayEmbedded(values) {
		input = JSON.parse(values);
		output = JSON.parse('{}');
		if (!input.obj1.hasOwnProperty('orderlines')) {
		embed_obj = JSON.parse('{}');
		embed_obj.orderlineid = input.obj1.orderlineid;
		embed_obj.orderid = input.obj1.orderid;
		embed_obj.prod_id = input.obj1.prod_id;
		embed_obj.quantity = input.obj1.quantity;
		embed_obj.products = input.obj1.products;
		if (input.obj2 != null)
		output.orderlines = [embed_obj, input.obj2];
		else
		output.orderlines = [embed_obj];
		} else {
		output = input.obj1;
		if (input.obj2 != null)
		output.orderlines.push(input.obj2);
		}
		return JSON.stringify(output);
		}

Figura 6.6: FDUs geradas no processo de tradução do DAG *Orders* em comandos para converter instâncias RDB para instâncias no modelo NoSQL orientado a documentos.

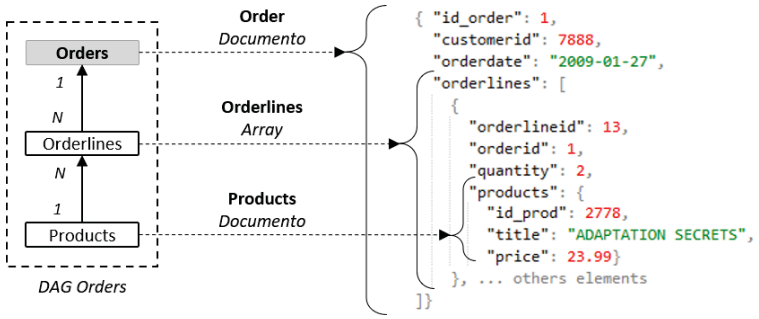


Figura 6.7: Resultado da conversão de uma instância de *Order* para NoSQL orientado a documentos.

6.2.2.2 *RDB para NoSQL Orientado a Família de Colunas*

Abaixo são apresentados os mapeamentos e comandos de transformação de dados gerados para converter instâncias do RDB de acordo com o DAG *Orders* da Figura 6.4:

$$m_{1.1} = (s(_{id\_order}), t(_{id\_order}), f_{casting});$$
$$m_{1.2} = (s(_{customerid}), t(_{customerid}), f_{casting});$$
$$m_{1.3} = (s(_{orderdate}), t(_{orderdate}), f_{casting});$$
$$m_{1.4} = (s(_{id\_prod,category,title,actor,price,special}), t(_{products}), f_{Js1});$$
$$cmd_1 = (((Products \bowtie Orderlines) \bowtie Orders), fMaps, MapReduce)$$

$$m_{2.1} = (s(_{id\_order}), t(_{rowkey}), f_{casting});$$
$$m_{2.2} = (s(_{products}), t(_{products}), f_{casting});$$
$$m_{2.3} = (s(_{id\_order,customerid,orderdate}), t(_{orders}), f_{Js2});$$
$$m_{2.4} = (s(_{orderlineid,orderid,prod\_id,quantity}), t(_{orderlines}), f_{Js3});$$
$$cmd_2 = ((Orderlines \bowtie Orders), fMaps, MapReduce)$$

O comando cmd_1 encapsula operações para transformar as entidades *Products* e *Orders* em objetos JSON e aninhar *Products* como uma família de colunas em *Orders*. O comando cmd_2 define o campo *rowkey* a partir do campo *id_order*, mantém o objeto *Products* criado a partir de

cmd_1 e cria as famílias de colunas *Orders* e *Orderlines*, encapsulando os campos das respectivas entidades em objetos JSON. O processo de criação de família de colunas como objetos JSON é encapsulado nas FDUs *product_manyNesting*, *order_oneNesting* e *orderline_manyNesting* (Figura 6.8). Essas FDUs são geradas automaticamente a partir dos vértices e arestas do DAG por meio da função *JScriptGenerator* (Algoritmo 3). O usuário pode modificar o código das FDUs para aplicar customizações antes de executar as transformações de dados.

Origem	Destino	FDU

id_prod,
category,
title,
actor,
price,
Special
	products	function product_manyNesting(values) {
		input = JSON.parse(values);
		output = JSON.parse('{}');
		if (!input.obj1.hasOwnProperty('products')) {
		key = input.obj1.id_prod;
		output.products = JSON.parse('{}');
		output.products[key + '_id_prod'] = input.obj1.id_prod;
		output.products[key + '_category'] = input.obj1.category;
		...
		} else {
		output.products = input.obj1.products;
		}
		if (input.obj2 != null) {
		key = input.obj2.id_prod;
		output.products[key + '_id_prod'] = input.obj2.id_prod;
		output.products[key + '_category'] = input.obj2.category;
		...
		}
		return JSON.stringify(output);
		}

id_order,
customerid,
orderdate
	orders	function order_oneNesting(values) {
		input = JSON.parse(values);
		output = JSON.parse('{}');
		if (!input.obj1.hasOwnProperty('orders')) {
		output.orders = JSON.parse('{}');
		output.orders.id_order = input.obj1.id_order;
		output.orders.customerid = input.obj1.customerid;
		output.orders.orderdate = input.obj1.orderdate;
		} else {
		output.orders = input.obj1.orders;
		}
		return JSON.stringify(output);
		}

orderlineid,
orderid,
prod_id,
quantity
	orderlines	function orderline_manyNesting(values) {
		input = JSON.parse(values);
		output = JSON.parse('{}');
		if (!input.obj1.hasOwnProperty('orderlines')) {
		key = input.obj1.orderlineid;
		output.orderlines = JSON.parse('{}');
		output.orderlines[key+'_orderlineid'] = input.obj1.orderlineid;
		output.orderlines[key+'_orderid'] = input.obj1.orderid;
		...
		} else {
		output.orderlines = input.obj1.orderlines;
		}
		if (input.obj2 != null) {
		key = input.obj2.orderlineid;
		output.orderlines[key+'_orderlineid'] = input.obj2.orderlineid;
		output.orderlines[key+'_orderid'] = input.obj2.orderid;
		...
		}
		return JSON.stringify(output);
		}	

Figura 6.8: FDUs geradas no processo de tradução do DAG *Orders* em comandos para converter instâncias RDB para instâncias no modelo NoSQL orientado a família de colunas.

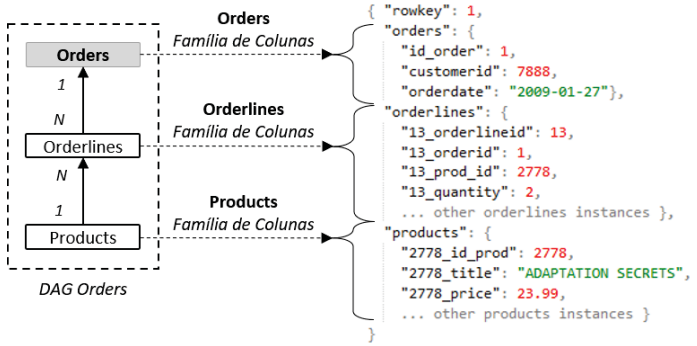


Figura 6.9: Resultado da conversão de uma instância de *Order* para NoSQL orientado a família de colunas.

A Figura 6.9 exibe uma instância de *Orders* depois da execução dos comandos. A instância de *Orders* é composta por campo *rowkey* e famílias de colunas *Orders*, *Orderlines* e *Products*. Para criar as famílias de colunas *Orderlines* e *Products*, os campos *orderlineid* e

id_prod foram usados como chave de aninhamento (*nesting_key*). A chave de aninhamento é usada para atribuir um qualificador de coluna único para cada campo da instância aninhada.

Após a execução dos experimentos de conversão foram realizadas verificações de consistência dos dados transformados. O objetivo foi verificar se todas as instâncias de dados do RDB foram convertidas para instâncias NoSQL sem perda de dados. A análise foi realizada manualmente por amostragem, em que foram comparadas instâncias das bases NoSQL em relação as instâncias da base relacional. Através da análise dos resultados foi possível constatar que a abordagem manteve a consistência entre a entrada e saída de dados.

6.2.3 Avaliação de Desempenho

O objetivo principal do *Experimento II* é avaliar a viabilidade do processo de conversão e migração de dados, deixando avaliações de desempenho detalhadas para trabalhos futuros. No entanto, o tempo de execução para converter os bancos de dados *RDB1*, *RDB2* e *RDB3* para NoSQL orientado a documento e NoSQL orientado a família de colunas foi mensurado. A Tabela 6.3 apresenta o tempo em segundos para o modelo orientado a documentos (*Doc*), modelo orientado a família de colunas (*Col*) e o número de instâncias (*Ins*) criadas, conforme os DAGs *Orders*, *Orderlines* e *Products*. É importante mencionar que o tempo mensurado reflete somente o tempo de conversão das instâncias, sem considerar tempo de persistência dos objetos JSON em disco ou no banco de dados NoSQL.

Tabela 6.3: Tempo de execução para converter os dados dos *RDB1*, *RDB2* e *RDB3*, conforme os DAGs da Figura 6.4, para NoSQL orientado a documentos e família de colunas.

DAG	**RDB1**			**RDB2**			**RDB3**		
	Doc	*Col*	*Ins*	*Doc*	*Col*	*Ins*	*Doc*	*Col*	*Ins*
Orders	32s	48s	12k	82s	138s	60k	140s	259s	120k
Orderlines	7s	15s	60k	24s	53s	301k	46s	105s	601k
Products	16s	29s	10k	79s	176s	50k	234s	597s	100k

Os resultados mostram que o tempo de execução aumenta, conforme aumenta o tamanho do RDB de entrada. Além disso, a conversão para o modelo orientado a família de colunas tem maior tempo de execução em comparação ao modelo orientado a documentos. Esse resultado está relacionado ao custo das operações de junção e número de chamadas a funções *MapReduce*, que é maior no processo de conversão para o modelo orientado a família de colunas.

6.3 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Neste capítulo foram apresentados dois experimentos realizados para verificar a viabilidade do *framework* Metamorfose e o processo de conversão e migração de bases relacionais para bases NoSQL orientadas a documentos e famílias de colunas.

O primeiro experimento apresentou os resultados obtidos sobre um cenário real de transformação de dados relacionais, publicados na trilha demo do SBBD'18 (Kuszera et al., 2018). O segundo experimento apresentou resultados que mostram a viabilidade do processo de conversão e migração para bases NoSQL. Ademais, os experimentos e resultados obtidos foram apresentados na conferência SAC'19 (Kuszera et al., 2019).

Durante o processo de conversão de RDB para NoSQL é possível definir diferentes DAGs (ou esquemas) para estruturar como os dados serão migrados. Essa flexibilidade torna

a escolha da estrutura adequada uma tarefa não trivial. No próximo capítulo será apresentado um conjunto de métricas e escores, que tem por objetivo auxiliar no processo de avaliação e comparação de esquemas NoSQL orientados a documentos, antes de realizar a migração de dados através do *framework* Metamorfose.

7 MÉTRICAS BASEADAS EM CONSULTAS (QBM)

Este capítulo apresenta o conjunto de métricas baseada em consultas (QBM, do inglês *Query-Based Metrics*)[1] e escores, referente às etapas **2. Cálculo de Métricas e Escores** e **3. Avaliação e Seleção de Esquemas** definidas no Capítulo 4. Foram definidas seis métricas para medir a cobertura que um esquema NoSQL documento tem em relação a um conjunto de consultas. Na sequência é apresentado um procedimento para conduzir a avaliação de esquemas, em que são usados escores para calcular a cobertura fornecida pelo esquema, por métrica e consulta. Também é apresentada a ferramenta *QBMetrics*, desenvolvida para dar suporte ao uso das métricas e escores no processo de avaliação e comparação de esquemas. Por último são apresentadas diretrizes para implementar as consultas conforme estrutura do esquema, tendo por objetivo padronizar a implementação das consultas.

7.1 CENÁRIO ILUSTRATIVO

Para ilustrar o uso das métricas, no restante deste capítulo serão utilizados como entradas o esquema da Figura 7.1 e as cinco consultas da Figura 7.2. Ambos foram apresentados no Capítulo 4 e estão sendo reapresentados aqui para facilidade de leitura.

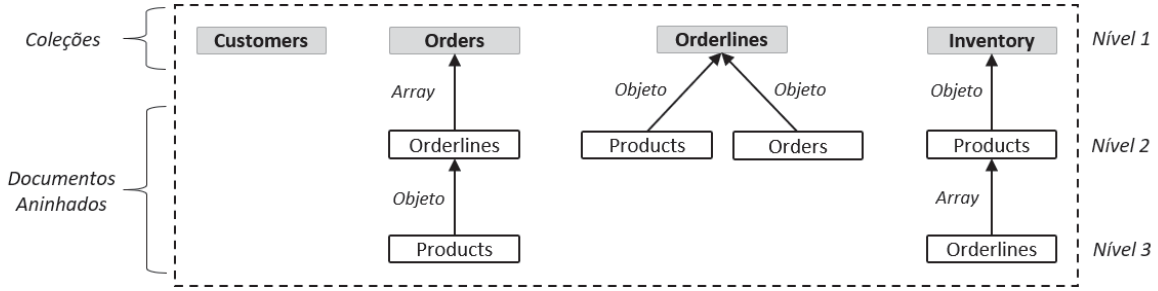


Figura 7.1: Esquema NoSQL orientado a documentos representado como um conjunto de DAGs.

7.2 DEFINIÇÃO DE MÉTRICAS PARA AVALIAÇÃO DE ESQUEMA

As métricas medem a cobertura que um esquema NoSQL fornece para o padrão de acesso da consulta. No contexto desta tese, o padrão de acesso é representado pelo conjunto de caminhos do DAG da consulta, e a cobertura é definida como a correspondência entre vértices e arestas do DAG do esquema e do DAG da consulta. Na Figura 7.2 todas as consultas têm apenas um caminho, mas é possível consultas com dois ou mais caminhos (exemplos no próximo Capítulo). Para calcular a cobertura são considerados os seguintes tipos de caminhos:

[1]Parte do conteúdo deste capítulo foi publicado em:

- Kuszera, E. M., Peres, L. M. e Didonet Del Fabro, M. (2020). Query-Based Metrics for Evaluating and Comparing Document Schemas. Em Dustdar, S., Yu, E., Salinesi, C., Rieu, D. e Pant, V., editores, Advanced Information Systems Engineering, páginas 530–545, Cham. Springer International Publishing.
- Kuszera, E. M., Peres, L. M. e Didonet Del Fabro, M. (2020). QBMetrics: A Tool for Evaluating and Comparing Document Schemas. Em Herbaut, N. e La Rosa, M., editores, Advanced Information Systems Engineering - CAiSE Forum, páginas 77–85, Cham. Springer International Publishing.

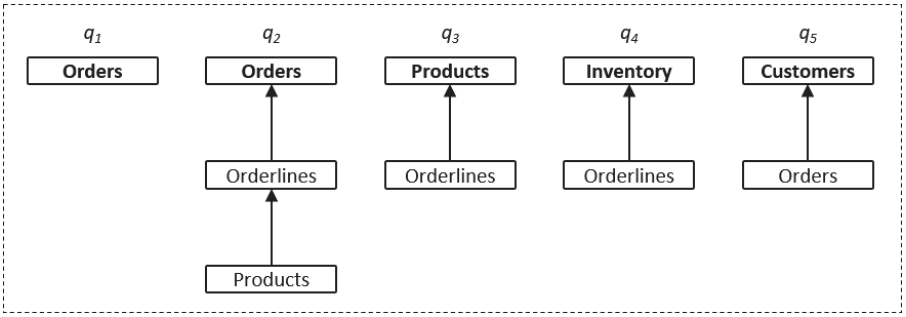


Figura 7.2: Conjunto de consultas SQL representadas como DAGs.

- *path*: um *path* p é uma sequência de vértices $v_1, v_2, ..., v_j$, tal que $(v_i, v_{i+1}) \in V_q$, $1 \leq i \leq j-1$, v_1 é o vértice raiz e v_j é o vértice folha do DAG. Essa sequência de vértices pode ser chamada de caminho entre vértice raiz e vértice folha.

- *sub path*: considerando um *path* $p = (v_1, v_2, ..., v_k)$ e, para qualquer i e j, tal que $1 \leq i \leq j \leq k$, um *subpath* de p é definido como $p_{ij} = (v_i, v_{i+1}, ..., v_j)$, na direção do vértice j para o vértice i.

- *indirect path*: considerando um *path* $p = (v_1, v_2, ..., v_k)$ e, para qualquer i, y e j, tal que $1 \leq i \leq y \leq j \leq k$, um *indirect path* relativo a p é definido como $p_{ind} = (v_i, v_{i+1}, ..., v_j)$, em que $\exists v_y \in p : v_y \notin p_{ind}$. Ou seja, um *indirect path* p_{ind} é aquele em que todos os seus vértices e arestas estão contidos em p, mas há vértices intermediários em p que separam um ou mais vértices de p_{ind}.

Para tornar as definições das métricas mais claras serão usados os seguintes termos: Q define o conjunto de consultas de entrada e $q \in Q$ uma determinada, V_q, V_s e V_c definem o conjunto de vértices de uma consulta, esquema e coleção, respectivamente. Os termos P_q, P_s e P_c definem o conjunto de caminhos de uma consulta, esquema e coleção, respectivamente. A seguir serão apresentadas as métricas e exemplos de uso, considerando o cenário ilustrativo apresentado na seção anterior.

7.2.1 *Direct Edge Coverage*

DirectEdge ou *DirEdge* (equação 7.1) mede a cobertura das arestas da consulta contra as arestas de uma coleção do esquema, considerando a direção das arestas ($a \rightarrow b$, por exemplo). E_{dq} e E_{dc} denotam o conjunto de arestas da consulta e da coleção de documentos, considerando a direção das arestas. A cobertura em relação ao esquema (equação 7.2) é o valor máximo encontrado ao aplicar a métrica *DirEdge* para cada coleção $c \in C$, tal que C é o conjunto de coleções do esquema. A função *Max* é uma função de alta-ordem que recebe um conjunto de coleções C, a consulta q e a função da métrica (*DirEdge*, por exemplo). *Max* aplica a função da métrica para todos os elementos da coleção C e consulta q. Essa função é usada na definição das demais métricas apresentadas neste capítulo.

$$DirEdge(c, q) = |(E_{dq} \cap E_{dc})|/|E_{dq}| \tag{7.1}$$

$$DirEdge(q) = Max(C, q, DirEdge) \tag{7.2}$$

Exemplo: $DirEdge(q_2)$ retorna 1.0, o que significa que há uma coleção no esquema que cobre todas as arestas da consulta, com equivalência de direção. $DirEdge(c_i, q_2)$ retorna a cobertura

de arestas, por coleção do esquema. As coleções $Orders$ e $Orderlines$ cobrem 100% e 50% das arestas de q_2, respectivamente.

7.2.2 *All Edge Coverage*

AllEdge (equação 7.3) mede a cobertura de arestas entre consulta e coleção de documentos, independente da direção da aresta ($a \rightarrow b$ e $a \leftarrow b$ são equivalentes, por exemplo). E_q e E_c denotam as arestas da consulta e coleção, respectivamente. A cobertura relativa ao esquema (equação 7.4) é o valor máximo encontrado ao aplicar *AllEdge* para cada coleção $c \in C$.

$$AllEdge(c, q) = |(E_q \cap E_c)|/|E_q| \tag{7.3}$$

$$AllEdge(q) = Max(C, q, AllEdge) \tag{7.4}$$

Exemplo: $AllEdge(q_2)$ retorna 1.0, o que significa que há uma coleção no esquema que cobre todas as arestas da consulta. $AllEdge(c_i, q_2)$ retorna a cobertura de arestas, por coleção do esquema. As coleções $Orders$ e $Orderlines$ cobrem 100% das arestas. No entanto, $Orders$ tem $DirEdge = 1.0$ e $Orderlines$ tem $DirEdge = 0.5$, ou seja, $Orderlines$ tem uma aresta invertida. É interessante usar $AllEdge$ em conjunto com $DirEdge$ para verificar se o esquema está invertido em relação ao padrão de acesso da consulta.

7.2.3 *Path Coverage*

Path mede a cobertura dos caminhos da consulta em relação aos caminhos da coleção de documentos. Uma consulta pode ter um ou mais caminhos, sendo que todos são considerados no cálculo da cobertura da métrica $Path$. Na equação (7.5) $Path$ mede a cobertura dos caminhos da consulta (P_q) em relação aos caminhos da coleção (P_c). Na equação (7.6) $Path$ mede a cobertura para todo o esquema, encontrando o valor máximo ao aplicar a métrica para cada coleção $c \in C$.

$$Path(c, q) = |(P_q \cap P_c)|/|P_q| \tag{7.5}$$

$$Path(q) = Max(C, q, Path) \tag{7.6}$$

Exemplo: $Path(q_2)$ retorna 1.0, o que significa que há uma coleção no esquema (coleção $Orders$) que cobre todos os caminhos da respectiva consulta. Para as demais consultas $Path$ retorna 0.0. A métrica $Path$ identifica esquemas que correspondem exatamente ao padrão da consulta.

7.2.4 *Sub Path Coverage*

SubPath verifica se os caminhos da consulta estão presentes no esquema como sub-caminhos. Para encontrar sub-caminhos no esquema foi definida a função $existSubPath$, que recebe como parâmetros um caminho da consulta $qp \in P_q$ e o conjunto de caminhos da coleção P_c. Se há um sub-caminho na coleção que corresponda com o caminho da consulta, a função retorna 1, caso contrário retorna 0. Na equação (7.7) é calculada a cobertura de sub-caminhos

para uma determinada coleção, e na equação (7.8) é calculada a cobertura para todo o esquema, aplicando $SubPath$ para todas as coleções $c \in C$, retornando o valor máximo encontrado.

$$existSubPath(qp, P_c) = \begin{cases} 1 & \text{found } qp \text{ as subpath in } P_c \\ 0 & \text{not found } qp \text{ as subpath in } P_c \end{cases}$$

$$SubPath(c, q) = \frac{\sum_{i=1}^{|P_q|} existSubPath(qp_i, P_c)}{|P_q|} \tag{7.7}$$

$$SubPath(q) = Max(C, q, SubPath) \tag{7.8}$$

Exemplo: $SubPath(q_1)$ retorna 1.0, o que significa que há uma coleção no esquema que cobre os caminhos de q_1 como sub-caminhos. $SubPath(c_i, q_1)$ retorna a cobertura de sub-caminhos, por coleção do esquema. As coleções $Orders$ e $Orderlines$ cobrem 100% dos caminhos de q_1 como sub-caminhos. No entanto, em $Orders$ a profundidade da localização do sub-caminho no esquema é $depth = 1$ e em $Orderlines$ é $depth = 2$. Neste caso, $Orders$ é melhor ranqueada pela métrica $SubPath$. As consultas q_2 e q_3 também são cobertas como sub-caminhos no esquema ($SubPath = 1.0$).

7.2.5 *Indirect Path Coverage*

IndirectPath ou *IndPath* verifica se os caminhos da consulta estão presentes no esquema como caminhos indiretos (ver definição no início da seção 7.2). Para encontrar caminhos indiretos no esquema foi definida a função $existIndPath$, que recebe como parâmetros um caminho da consulta $qp \in P_q$ e o conjunto de caminhos da coleção P_c. Se há um caminho indireto na coleção que corresponda com o caminho da consulta, a função retorna 1, caso contrário retorna 0. Na equação (7.9) é calculada a cobertura de caminhos indiretos para uma determinada coleção, e na equação (7.10) para todo o esquema, aplicando $IndPath$ para cada coleção $c \in C$. O maior valor encontrado é retornado como a cobertura para o esquema.

$$existIndPath(qp, P_c) = \begin{cases} 1 & \text{found } qp \text{ as an indirect path in } P_c \\ 0 & \text{not found } qp \text{ as an indirect path in } P_c \end{cases}$$

$$IndPath(c, q) = \frac{\sum_{i=1}^{|P_q|} existIndPath(qp_i, P_c)}{|P_q|} \tag{7.9}$$

$$IndPath(q) = Max(C, q, IndPath) \tag{7.10}$$

Exemplo: $IndPath(q_4)$ retorna 1.0, o que significa que há uma coleção no esquema em que os caminhos de q_4 são cobertos como caminhos indiretos. $IndPath(c_i, q_4)$ retorna a cobertura, por coleção do esquema. Neste caso, a coleção $Inventory$ cobre 100% do caminho de q_4 como caminho indireto. O caminho indireto é devido ao vértice $Products$ na coleção $Inventory$, que está entre os vértices $Inventory$ e $Orderlines$ da consulta q_4.

7.2.6 *Required Collections Coverage*

Required Collections ou *ReqColls* (equação 7.11) retorna o menor número de coleções necessárias para responder uma determinada consulta. A função $createCollectionPaths(q)$

retorna um conjunto de caminhos composto por coleções que têm as entidades necessárias para responder a consulta.

$$ReqColls(q) = min(createCollectionPaths(q)) \tag{7.11}$$

Exemplo: $ReqColls(q_5)$ retorna 2, indicando que é necessário realizar a junção de documentos de duas coleções para responder q_5, sendo as coleções $Customers$ e $Orders$. As demais consultas têm $ReqColls = 1$, indicando que é possível responder a respectiva consulta por meio de uma coleção de documentos do esquema.

Esta seção apresentou um conjunto de métricas para medir a cobertura fornecida pelo esquema em relação a uma determinada consulta. Na próxima seção será apresentado como as métricas podem ser usadas de forma isolada ou combinada para medir a cobertura de todo o conjunto de consultas em relação ao esquema.

7.3 AVALIAÇÃO DE ESQUEMAS USANDO AS MÉTRICAS

Esta seção apresenta um procedimento para conduzir a avaliação de esquemas por meio das métricas. Para tal, foram definidos dois tipos de escores. O primeiro define um escore por consulta e é chamado de $QScore$ (do inglês *Query Score*). Ele permite combinar uma ou mais métricas relacionadas e definir pesos para priorizar a importância de métricas específicas. O segundo é chamado de $SScore$ (do inglês *Schema Score*) e define um escore por métrica, considerando todas as consultas do conjunto Q. Os resultados são usados para ranquear o esquema de entrada. $QScore$ e $SScore$ serão explicados nas seções seguintes.

7.3.1 *Query Score*

Query Score ($QScore$) é calculado para uma determinada métrica e consulta q_i. Produz um único valor por métrica, ou um valor que combina métricas relacionadas. O $QScore$ para as métricas *DirEdge*, *AllEdge* e *ReqColls* é o mesmo valor retornado pela função da métrica:

$$QScore(DirEdge, q_i) = DirEdge(q_i) \tag{7.12}$$
$$QScore(AllEdge, q_i) = AllEdge(q_i) \tag{7.13}$$
$$QScore(ReqColls, q_i) = ReqColls(q_i) \tag{7.14}$$

No entanto, o $QScore$ para as métricas *Path*, *SubPath* e *IndPath* é um único valor, chamado de $Paths$. Esse escore retorna o maior valor entre as três métricas, levando em consideração a profundidade em que cada caminho foi localizado no esquema e um peso adicional, como definido a seguir:

$$path_v = Path(q_i) * w_p \tag{7.15}$$
$$subpath_v = (SubPath(q_i) * w_{sp})/depthSP(q_i) \tag{7.16}$$
$$indpath_v = (IndPath(q_i) * w_{ip})/depthIP(q_i) \tag{7.17}$$
$$QScore(Paths, q_i) = max(path_v, subpath_v, indpath_v) \tag{7.18}$$

O método apresentado para calcular $QScore(Paths)$ é inspirado nos resultados de (Gómez et al., 2016), na qual os autores declaram que a profundidade dos dados requeridos na coleção de documentos e a necessidade de acessar dados armazenados em diferentes níveis da coleção produzem impacto negativo na execução da consulta.

Os pesos w_p, w_{sp} e w_{ip} são usados para configurar a prioridade entre as métricas *Path*, *SubPath* e *IndPath*. A definição de pesos permite priorizar certo tipo de cobertura durante o processo de avaliação de esquemas. Por exemplo, a métrica *Path* denota a correspondência exata entre caminho da consulta e caminho do esquema. Uma possibilidade é configurar w_p com valor maior, seguido de valores menores para w_{sp} e w_{ip}. Desta forma, esquemas com correspondência exata são priorizados. As funções $depthSP()$ e $depthIP()$ retornam a menor profundidade em que foi localizado um caminho da consulta como um sub-caminho ou como um caminho indireto no esquema. A profundidade é calculada com base na localização do vértice raiz da consulta em relação ao esquema, e é usada para penalizar o valor final da métrica.

A expressão final (7.18) para calcular o $QScore$ aplica a função $max()$ sobre os valores das variáveis $path_v$, $subpath_v$ e $indpath_v$ (usadas apenas para melhorar a legibilidade do texto). Esquemas NoSQL podem apresentar redundância de dados, em que um caminho de consulta pode ser encontrado simultaneamente como um *Path*, *SubPath* e *IndPath* no mesmo esquema. Neste caso, a função $max()$ é usada para retornar a maior cobertura encontrada considerando as três métricas. Então, ao definir pesos distintos para as métricas e usar a profundidade do caminho é possível priorizar um tipo particular de cobertura para avaliar os esquemas.

7.3.2 *Schema Score*

Schema Score ($SScore$) é calculado para uma determinada métrica mt, esquema s e conjunto de consultas Q. O cálculo consiste na soma dos valores de $QScore(mt, q_i)$, tal que cada consulta q_i tem um peso específico w_i, e a soma de todos w_i é igual a 1. $ReqColls$ é um caso diferente, sendo explicado adiante. Seguindo a mesma ideia de $QScore$, há um único valor para *Path*, *SubPath*, e *IndPath*, que é a soma dos seus respectivos $QScores$. O cálculo do $SScore$ é definido a seguir:

$$SScore(mt, Q) = \sum_{i=1}^{|Q|} QScore(mt, q_i) * w_i \tag{7.19}$$

$SScore$ para a métrica $ReqColls$ é uma taxa entre o número de consultas e o número de coleções de documentos requeridas para respondê-las. Um esquema que responde cada consulta com somente uma coleção tem $SScore(ReqColls) = 1$. Esse número diminui quando o número de coleções requeridas aumenta. NC é o número de coleções requeridas para responder todas as consultas de Q, sendo calculado como a soma de $QScore(ReqColls, q_i)$. A expressão para calcular $SScore$ para a métrica $ReqColls$ é baseada na métrica de minimalidade de esquemas (Cherfi et al., 2003), sendo definida da seguinte forma:

$$NC = \sum_{i=1}^{|Q|} QScore(ReqColls, q_i)$$

$$SScore(ReqColls, Q) = \frac{|Q|}{NC} \tag{7.20}$$

Exemplo: a Tabela 7.1 mostra o $QScore$ e $SScore$ para as métricas $Paths$, $DirEdge$, $AllEdge$ e $ReqColls$ calculados sobre o esquema da Figura 7.1. Para calcular os escores foram atribuídos pesos iguais para todas as consultas, totalizando 1. Os seguintes pesos foram atribuídos para as métricas $Path$, $SubPath$ e $IndPath$: $w_p = 1.0$, $w_{sp} = 0.7$ e $w_{ip} = 0.5$. Os pesos representam a prioridade entre as métricas, de forma que a cobertura $path$ é preferível em relação as demais.

Outros valores para os pesos podem ser definidos, dependendo dos requisitos definidos para avaliar o esquema.

Tabela 7.1: $QScore$ e $SScore$ das consultas $q_1 - q_5$ calculados sobre o esquema da Figura 7.1.

Query	Paths	DirEdge	AllEdge	ReqColls
q1	0.7	0.0	0.0	1
q2	1.0	1.0	1.0	1
q3	0.35	1.0	1.0	1
q4	0.5	0.0	0.0	1
q5	0.0	0.0	0.0	2
SScore	0.51	0.40	0.40	0.83

Esse procedimento para calcular $QScore$ e $SScore$ é usado para avaliar e comparar esquemas NoSQL em relação ao padrão de acesso da aplicação. $QScore$ mostra a cobertura fornecida pelo esquema para cada métrica e consulta, permitindo identificar quais consultas requerem maior atenção ou não são cobertas pelo esquema. O campo $SScore$ fornece uma visão geral da cobertura fornecida pelo esquema em relação ao conjunto de consultas. Por meio da definição de pesos para métricas e consultas, é possível configurar diferentes cenários de avaliação, e usar os resultados obtidos para ranquear esquemas candidatos em relação ao padrão de acesso da aplicação. Esse procedimento será usado na execução dos experimentos apresentados no Capítulo 8.

7.4 FERRAMENTA *QBMETRICS*

Para dar suporte ao processo de avaliação de esquemas NoSQL por meio das métricas e escores foi desenvolvida a ferramenta *QBMetrics* (Kuszera et al., 2020a). A *QBMetrics*[2] fornece suporte para as seguintes etapas da abordagem de conversão apresentada nesta tese:

1. Especificação de Esquemas e Consultas,
2. Cálculo de Métricas e Escores,
3. Avaliação e Seleção de Esquema.

A Figura 7.3 exibe a interface gráfica do *QBMetrics*. O fluxo de execução da ferramenta é composto por quatro etapas. Em **(A)** são fornecidos os parâmetros de conexão com o RDB de entrada. Na versão atual são suportados os bancos de dados *Postgre*s e *MySQL*. Em **(B)** são definidos um ou mais esquemas NoSQL, com base nos metadados do RDB de entrada. Em **(C)** são definidas as consultas que representam o padrão de acesso da aplicação. Em **(D)** as métricas e escores são calculados. Após o processo de avaliação e seleção do esquema NoSQL **(D)** por parte do usuário, a ferramenta permite exportar o esquema selecionado para o *framework* Metamorfose executar a migração dos dados.

[2]A ferramenta *QBMetrics* está disponível para download em: `https://github.com/evandrokuszera/nosql-query-based-metrics`

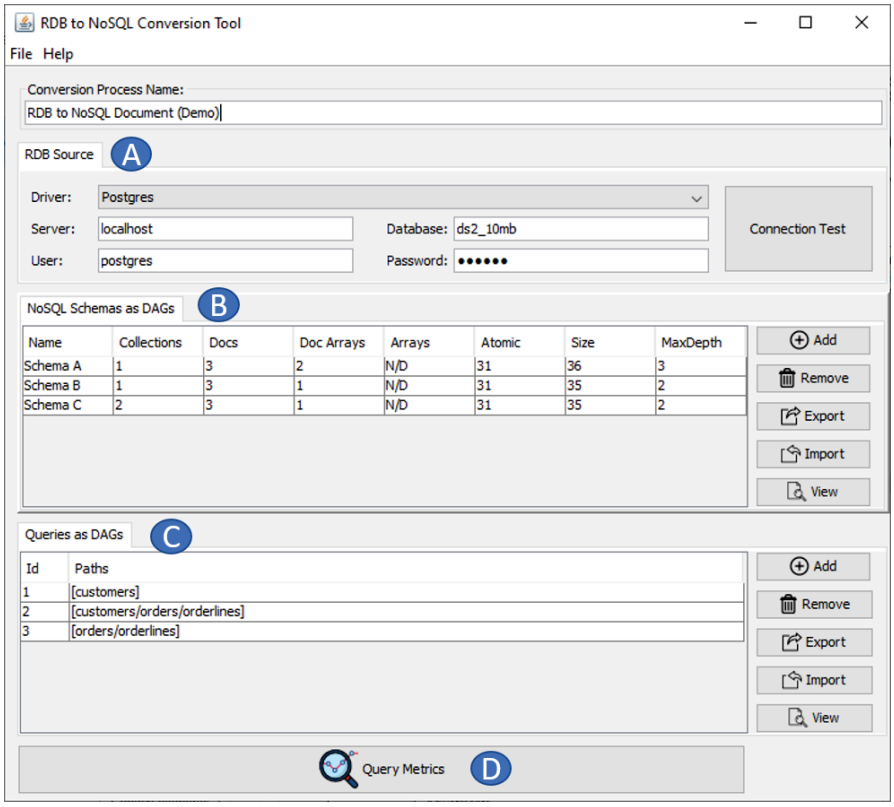


Figura 7.3: Interface gráfica da ferramenta *QBMetrics*.

7.4.1 Definindo Esquemas e Consultas

QMetrics fornece uma interface gráfica para definir os esquemas e consultas como DAGs, a partir dos metadados do RDB. No contexto de conversão de RDB para NoSQL orientado a documentos é possível definir um ou mais esquemas NoSQL candidatos. A Figura 7.4 exibe a interface gráfica para definir esquemas NoSQL, sendo possível adicionar novas coleções de documentos ou remover coleções existentes. Na figura de exemplo é exibido um esquema denominado *schema C*, composto pelas coleções *Customers* e *Orders*.

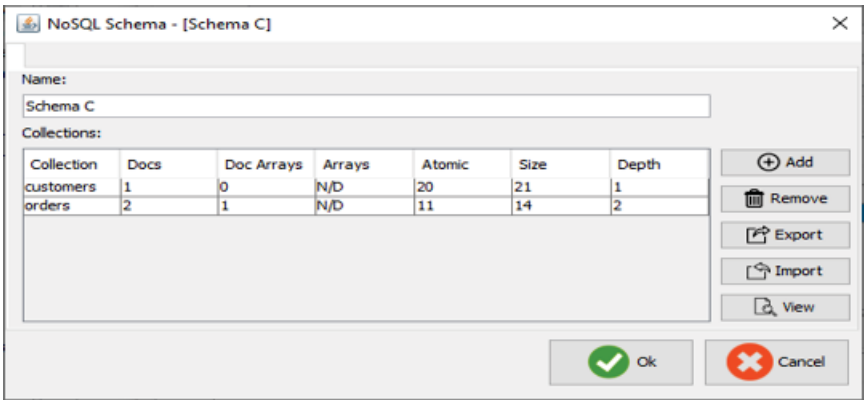


Figura 7.4: Interface gráfica para definir esquemas na ferramenta *QBMetrics*.

A Figura 7.5 exibe a interface gráfica para definir uma coleção de documentos para um determinado esquema NoSQL (referente a etapa **(B)** da Figura 7.3). Dois passos são necessários para definir uma coleção de documentos:

- *Step 1 - Add vertices*: o usuário seleciona os vértices (tabelas RDB) para compor o DAG. A lista de tabelas do RDB é carregada automaticamente pelo *QBMetrics*, com base nos metadados do banco de dados. Na Figura 7.5 os vértices *Orders* e *Orderlines* são selecionados.
- *Step 2 - Add edges*: o usuário adiciona arestas entre os vértices para estabelecer a ordem de aninhamento entre as entidades. Ao selecionar o vértice de origem (*source vertex*) a ferramenta automaticamente busca por possíveis vértices destino (*target vertex*), conforme metadados do RDB. Por exemplo, ao selecionar *Orderlines* como origem, a *QBMetrics* automaticamente sugere *Orders* como destino.

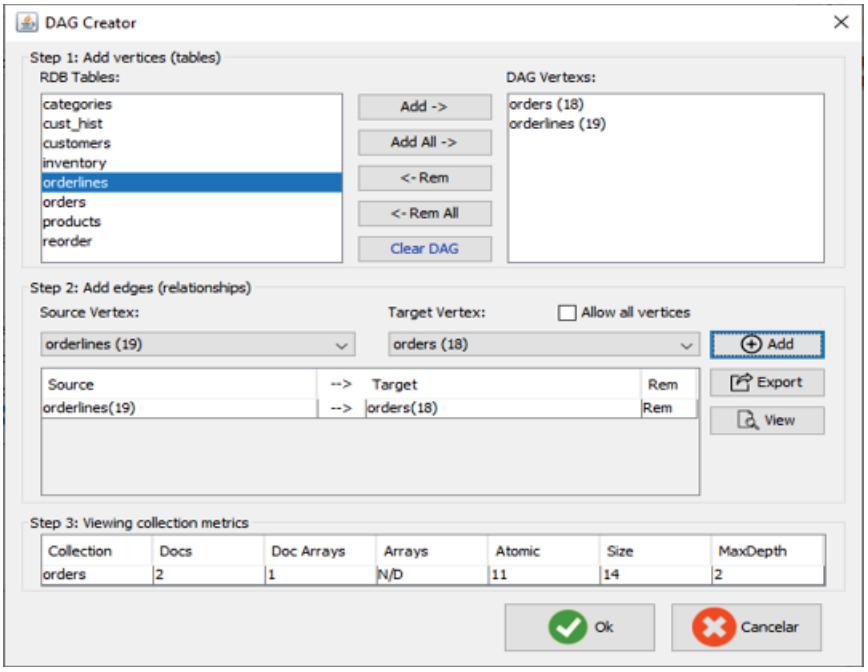


Figura 7.5: Interface gráfica para definir coleções de documentos ou consultas como DAGs, na ferramenta *QBMetrics*.

O processo para definir consultas (etapa **(C)** da Figura 7.3) é similar ao processo para definir coleções de documentos. A mesma interface gráfica apresentada na Figura 7.5 é usada para definir as consultas.

7.4.2 Visualizando Métricas e Escores

A Figura 7.6 exibe os resultados dos cálculos das métricas e escores, referente a etapa **(D)**. É possível filtrar os resultados por consulta, por métrica, por esquema e por tipo de escore. Além disso, nesta etapa é possível definir pesos para as métricas e consultas, e recalcular os resultados por meio da interface gráfica da ferramenta.

O objetivo do *QBMetrics* é fornecer suporte às etapas da abordagem de conversão de RDB para NoSQL, e auxiliar o usuário no processo de avaliação, comparação e seleção de esquemas candidatos, antes de realizar a migração de dados. Ademais, a ferramenta permite persistir o processo de conversão para uso posterior ou para auditoria.

7.5 ESTRATÉGIAS PARA IMPLEMENTAÇÃO DE CONSULTAS

A abordagem de conversão e migração apresentada neste trabalho não define um mecanismo automático de tradução das consultas SQL para a linguagem de consulta nativa

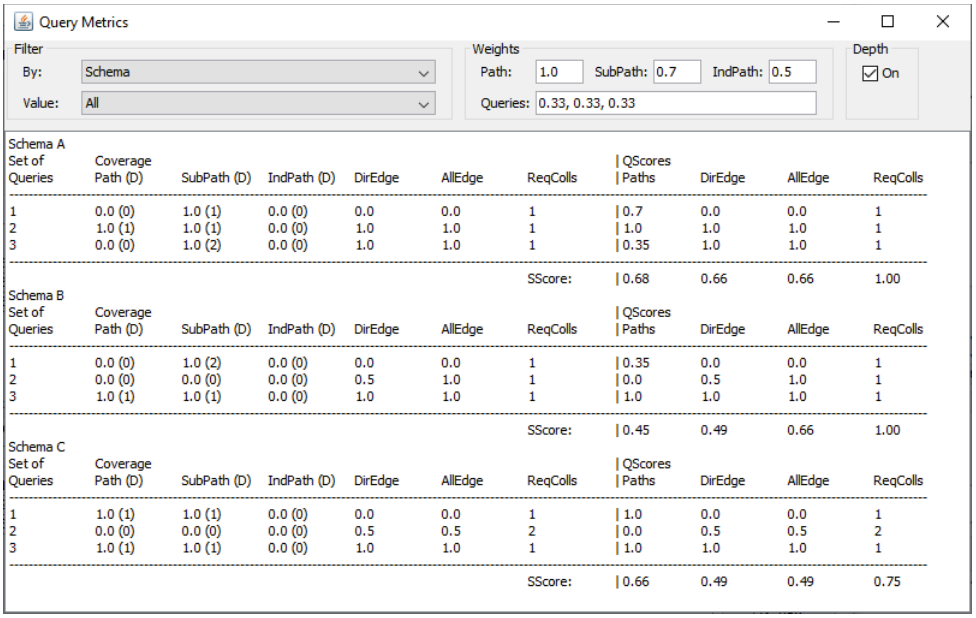


Figura 7.6: Interface gráfica para exibir os resultados das métricas e escores produzidos por meio do *QBMetrics*.

da base NoSQL. No entanto, para analisar se há relação entre métricas, escores e esforço de implementação de consultas foram definidas diretrizes de implementação. Essas diretrizes têm como objetivo padronizar o processo de implementação, assumindo que o DAG da consulta representa o formato final em que os dados recuperados da base NoSQL são retornados à aplicação.

As diretrizes de implementação foram estabelecidas considerando o *framework Aggregation Pipeline*, disponível no MongoDB versão 4.2. Esse *framework* permite expressar uma consulta como um *pipeline*, composto por um conjunto de estágios que representam operações sobre os documentos. Em cada estágio é possível expressar operações de seleção, agrupamento e projeção sobre os documentos. Além disso, há um tipo de estágio para realizar a junção de documentos entre duas coleções (semelhante ao *JOIN* usado em SQL). O banco de dados MongoDB foi escolhido pois é um dos mais usados pela comunidade, dispondo de ampla documentação. Porém, é importante destacar que ainda não há uma linguagem padrão de consulta sobre bases de dados orientadas a documentos. Além disso, as diretrizes podem ser adaptadas para uso em outras bases NoSQL orientadas a documentos.

No processo de implementação da consulta, o número e ordem de estágios e operadores utilizados variam conforme estrutura do DAG e a localização do filtro da consulta no esquema. Na versão 4.2 do MongoDB *Aggregation Pipeline* há vinte e oito tipos diferentes de estágios[3]. No entanto, as diretrizes de implementação são baseadas em sete estágios, apresentados na Tabela 7.2.

Para ilustrar a definição das diretrizes de implementação considere os esquemas e consultas apresentados na Figura 7.7. Os esquemas apresentam três formas de estruturar as entidades *A*, *B* e *C*. No esquema s_1 é usada ordem de aninhamento *1-N* entre as entidades *A* e *B*, enquanto no esquema s_2 a ordem usada é *N-1*. Para ambos os esquemas, a entidade *C* não está aninhada com outras entidades. De forma diferente, no esquema s_3 as entidades não estão aninhadas, sendo necessário realizar a junção de documentos entre as coleções para responder as consultas.

[3]https://docs.mongodb.com/manual/meta/aggregation-quick-reference/

Tabela 7.2: Conjunto de estágios usados para implementar as consultas.

Nome	Descrição
AddField	Remodela cada documento que passa pelo estágio, especificamente adicionando novos campos aos documentos de saída.
Group	Agrupa documentos de entrada de acordo com um identificador especificado. Consome todos os documentos de entrada e produz um documento por cada grupo distinto, contendo identificador e campos acumulados.
Lookup	Executa um *left outer join* com outra coleção no mesmo banco de dados, com objetivo de juntar documentos associados para processamento.
Match	Filtra o fluxo de documentos para permitir que apenas documentos correspondentes passem sem modificação para o próximo estágio do *pipeline*.
Project	Remodela cada documento no fluxo, adicionando novos campos ou removendo campos existentes. Para cada documento de entrada, gera um documento de saída.
ReplaceRoot	Substitui o documento de entrada por um dos seus documentos embutidos. O documento embutido é promovido como documento de nível superior.
Unwind	Desconstrói um campo de *array* de documentos de entrada para produzir um documento para cada elemento. Para cada documento de entrada, gera *n* documentos em que *n* é o número de elementos do *array* e pode ser zero para um *array* vazio.

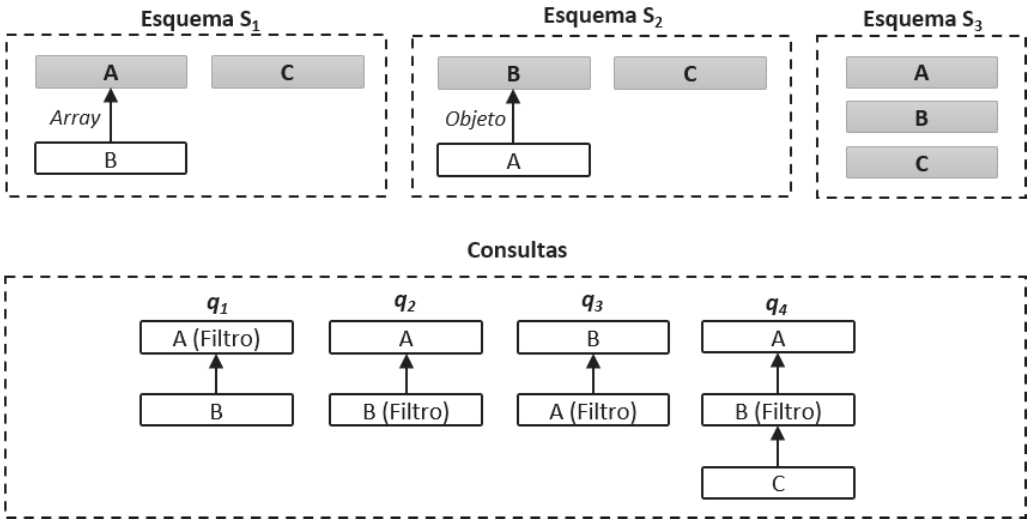


Figura 7.7: Esquemas e consultas usados para ilustrar as diretrizes de implementação de consultas.

Para implementar as consultas sobre cada esquema são necessárias operações para recuperar, filtrar e transformar os documentos do esquema, conforme a estrutura do DAG da consulta. Considere que (q_i, s_i) denota a implementação da consulta q_i sobre o esquema s_i. Em (q_1, s_1) são necessárias operações de recuperação e filtro dos documentos da coleção *A*. Não há necessidade de formatação dos dados retornados devido a semelhança estrutural entre os DAGs da consulta e esquema. Em (q_1, s_2) são necessárias operações de recuperação, filtro e transformação dos documentos da coleção *B* conforme estrutura do DAG da consulta. Para

(q_1, s_3) é necessário recuperar e filtrar os documento de A, realizar a junção com B, aninhando os documentos de B em A.

Dependendo da diferença estrutural entre DAG da consulta e DAG do esquema são necessários operadores para agrupar ou desagrupar os documentos do esquema, antes de executar operações de formatação. O operador *$group* é usado para aninhar documentos como *arrays* de documentos embutidos e o operador *$unwind* é usado para desagrupar *arrays* de documentos embutidos. A implementação (q_1, s_2) é um exemplo em que são necessárias operações de agrupamento. Primeiro são recuperados os documentos que correspondem ao filtro da consulta, na sequência as instâncias de B são agrupadas (*$group*) e aninhadas em A como um *array* de documentos embutidos. Outro exemplo é (q_3, s_1), em que após a operação de filtro é necessário desagrupar as instâncias de B (*$unwind*), para depois agrupar as instâncias de A e aninhá-las como um objeto embutido em B.

Outro aspecto que afeta a implementação da consulta é a localização do filtro em relação ao esquema. Quando o filtro incide sobre campo de *array* de documentos embutidos são necessárias operações adicionais para garantir que somente os dados relacionados ao filtro da consulta serão retornados. Essa situação ocorre para (q_2, s_1), em que o filtro da consulta incide sobre B, que é um *array* de documentos embutidos.

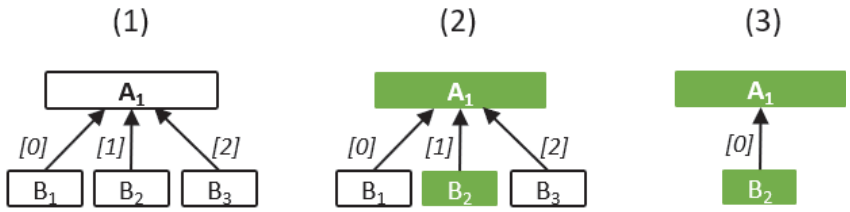


Figura 7.8: Impacto da localização do filtro da consulta em campo *array* de documentos embutidos no esquema.

Para ilustrar a implementação de (q_2,s_1) considere a Figura 7.8. Em (1) é exibida uma representação do documento A_1, armazenado na coleção A. Considerando que o filtro de q_2 corresponda ao elemento B_2, então o documento A_1 é retornado do banco de dados na etapa de filtro (2). No entanto, os elementos B_1 e B_3 não correspondem ao filtro da consulta e devem ser removidos (3) do documento, antes do resultado ser enviado para a aplicação. Para implementar essa consulta são necessárias as seguintes operações: filtrar os documentos de A, depois desagrupar (*$unwind*) o *array* de documentos embutidos (B, neste caso) para ter acesso aos elementos de forma individual, depois filtrar novamente os documentos de interesse e reagrupar (*$group*) os documentos conforme DAG da consulta. Como resultado serão retornados à aplicação somente os dados que correspondem ao filtro da consulta. Esse exemplo mostra como a localização do filtro tem impacto na implementação da consulta, mesmo quando há correspondência estrutural entre o DAG da consulta e DAG do esquema.

Com base nos aspectos apresentados acima são definidas diretrizes para uniformizar a implementação das consultas. Além de disso, as diretrizes constituem boas práticas de implementação e um passo inicial para automatizar a implementação das consultas em trabalhos futuros. Abaixo as diretrizes são definidas:

1. O DAG da consulta representa a estrutura na qual os documentos devem estar formatados após serem retornados do banco de dados.

2. A implementação da consulta deve iniciar por meio do operador *$match*, operador de filtro do *pipeline*. O objetivo é reduzir o número de documentos encaminhados para os demais estágios do *pipeline*.

3. Quando o filtro da consulta está localizado em campo *array* de documentos embutidos é necessário desagrupar (*$unwind*) os documentos, re-filtrar (*$match*) os documentos para descartar elementos relacionados com a raiz do documento, mas não relacionados com o filtro da consulta, para depois aplicar as demais diretrizes abaixo.

4. Na sequência, a ordem de implementação da consulta é de baixo (vértice folha) para cima (vértice raiz). Por exemplo, para implementar (q_3, s_1) primeiro são recuperadas as instâncias de A que correspondem ao filtro, e depois são executadas operações para aninhar A em B, seguindo estrutura do DAG da consulta.

5. Operações de projeção sobre os campos dos documentos (*$project*) são empurradas para o final do *pipeline*, salvo quando necessário executá-las em estágios intermediários.

6. Quando a consulta relaciona documentos de duas ou mais coleções:

   (a) A implementação do *pipeline* deve iniciar na coleção onde está localizado o filtro da consulta. Essa estratégia visa reduzir o número de documentos antes de realizar a operação *$lookup* (similar ao JOIN do SQL).

   (b) A ordem de execução de aninhamentos segue a diretriz 4. Por exemplo, considerando (q_4, s_3), a implementação da consulta deve iniciar com o operador *$match* sobre B, depois C é aninhado em B e, por último, $B(C)$ é aninhado em A.

   (c) Quando o ponto de junção está localizado em campo de *array* de documentos embutidos é necessário, primeiro, desaninhar os documentos para depois fazer a junção. Por exemplo, para implementar (q_4, s_1) é preciso desaninhar (*$unwind*) B, para depois aninhar (*$group*) C em B e, $B(C)$ em A.

As diretrizes de implementação serão usadas nos experimentos apresentados no Capítulo 8. O objetivo é padronizar a implementação das consultas e avaliar a relação entre métricas, escores e esforço de implementação de consultas para diferentes esquemas.

7.6 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Neste capítulo foram apresentadas as métricas baseadas em consultas (QBM) que são usadas para avaliar e comparar esquemas NoSQL em relação as consultas da aplicação. Ambos, esquemas e consultas são representados por meio de DAGs, permitindo calcular a cobertura que um esquema NoSQL fornece para as consultas. O DAG é usado para definir o padrão de acesso da consulta e o formato (ou estrutura) dos documentos retornados pelo banco de dados.

Foram definidas seis métricas: $Path$, $SubPath$ e $IndPath$ medem a cobertura de caminhos, $DirEdge$ e $AllEdge$ medem a cobertura de arestas, e $ReqColls$ mede o número de coleções necessárias para responder a consulta. Um procedimento para avaliar esquemas por meio das métricas foi apresentado, definindo dois escores: $QScore$ e $SScore$. Através desses escores o usuário pode avaliar e comparar esquemas candidatos antes da migração dos dados. Ademais, a definição das métricas e escores foram apresentados na conferência CAiSE'20 (Kuszera et al., 2020b). Também foi apresentada a ferramenta *QBMetrics*, desenvolvida para dar suporte ao processo de definição de esquemas e consultas, cálculo de métricas e escores, e para auxiliar o usuário na seleção de esquemas. Ademais, a ferramenta *QBMetrics* foi apresentada no fórum da conferência CAiSE'20 (Kuszera et al., 2020a).

Por último foram apresentadas diretrizes para implementar as consultas com base na estrutura do DAG, considerando como os documentos estão relacionados no esquema NoSQL.

Essas diretrizes foram identificadas durante o processo implementação das consultas e constituem um conjunto de boas práticas, além de padronizar o processo de implementação. No futuro pretende-se usar essas diretrizes para automatizar a implementação das consultas.

Ademais, as métricas apresentam um ponto de partida efetivo para escolher entre várias opções de esquemas NoSQL, em que o usuário pode priorizar consultas e métricas por meio da atribuição de pesos. No entanto, as métricas não são destinadas para avaliar o tempo de execução das consultas, pois não são consideradas informações sobre o tamanho dos documentos e cardinalidade das coleções e dos relacionamentos entre documentos. Métricas estruturais (Gómez et al., 2018) podem ser combinadas com as métricas apresentadas neste capítulo para fornecer mais subsídios para o processo de seleção de esquema (profundidade e redundância do esquema, por exemplo). Como trabalho futuro pretende-se explorar esse caminho.

No próximo capítulo serão apresentados os experimentos realizados para validar as métricas em um cenário de conversão de RDB para NoSQL orientado a documento.

8 EXPERIMENTOS: USO DAS MÉTRICAS BASEADAS EM CONSULTAS

Este capítulo apresenta experimentos realizados para validar o conjunto de métricas baseadas em consulta (QBM)[1]. Dois experimentos foram realizados em um cenário de conversão de RDB para NoSQL orientado a documentos, em que vários esquemas NoSQL candidatos são avaliados em relação a um conjunto de consultas previamente definido. O primeiro experimento aplica as métricas sobre esquemas e consultas com o objetivo de avaliar qual esquema melhor cobre o conjunto de consultas. O segundo experimento tem por objetivo avaliar o impacto da localização do filtro (ou predicado) da consulta no esquema NoSQL, em relação ao esforço de implementação e tempo de execução das consultas. Neste experimento as consultas são implementadas conforme as diretrizes propostas no capítulo anterior, em que são definidas estratégias para implementar as consultas conforme a estrutura do esquema.

8.1 EXPERIMENTO I: AVALIANDO DIFERENTES ESQUEMAS NOSQL ORIENTADOS A DOCUMENTOS

Este experimento tem por objetivo avaliar as métricas baseadas em consultas em um cenário de conversão de RDB para NoSQL documento. Para gerar os esquemas NoSQL candidatos foram selecionadas quatro abordagens de conversão de RDB para NoSQL a partir da literatura. O conjunto de consultas Q utilizado no experimento representa o padrão de acesso da aplicação sobre o RDB, sendo conhecido a priori. O objetivo principal do experimento é mostrar como usar as métricas e escores para auxiliar o usuário no processo de avaliar, comparar e selecionar o esquema NoSQL apropriado, antes de executar a migração de dados.

8.1.1 Criando Esquemas NoSQL Candidatos

Quatro abordagens de conversão RDB para NoSQL documento foram selecionadas (Stanescu et al., 2016; Zhao et al., 2014; Karnitis e Arnicans, 2015; Jia et al., 2016). Elas definem diferentes formas (regras) de converter dados relacionais para dados aninhados. As regras de transformação de cada abordagem são aplicadas sobre o RDB da Figura 8.1 para gerar um conjunto de esquemas NoSQL. Este RDB foi utilizado no Capítulo 5, mas é reapresentado aqui para facilidade de leitura. A Tabela 8.1 mostra as abordagens e os parâmetros de entrada usados para gerar os esquemas NoSQL. A natureza dos parâmetros de entrada é dependente da estratégia usada por cada abordagem. Foram criados quatro esquemas NoSQL, um para cada abordagem, nomeados com rótulos de S_a a S_d. A Figura 8.2 mostra a representação gráfica dos esquemas.

[1]Parte do conteúdo deste capítulo foi publicado em:

- Kuszera, E. M., Peres, L. M. e Didonet Del Fabro, M. (2020). Query-based metrics for evaluating and comparing document schemas. Em Dustdar, S., Yu, E., Salinesi, C., Rieu, D. e Pant, V., editores, Advanced Information Systems Engineering, páginas 530–545, Cham. Springer International Publishing.
- Kuszera, E. M., Peres, L. M. e Didonet Del Fabro, M. (2020). QBMetrics: A Tool for Evaluating and Comparing Document Schemas. Em Herbaut, N. e La Rosa, M., editores, Advanced Information Systems Engineering - CAiSE Forum, páginas 77–85, Cham. Springer International Publishing.

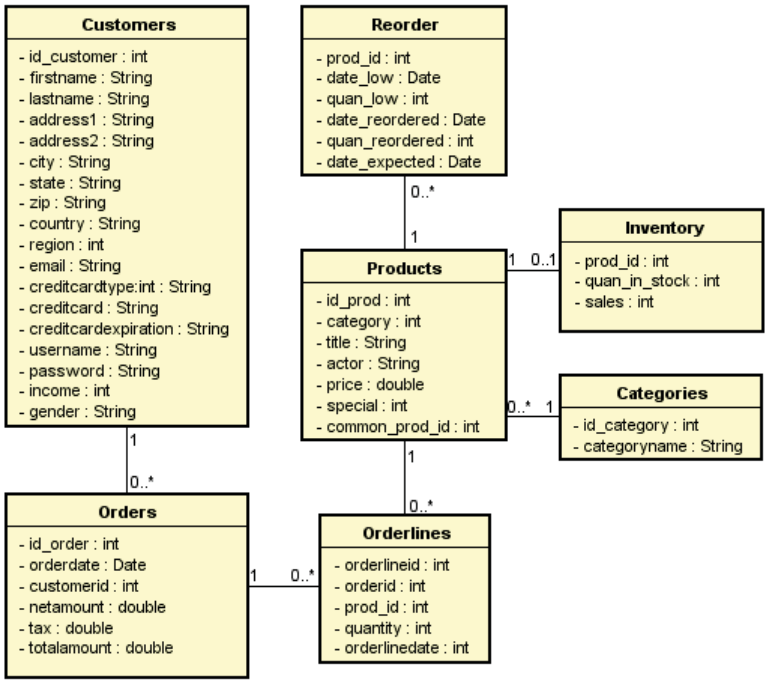


Figura 8.1: Diagrama Entidade-Relacionamento do banco de dados da aplicação *Dell DVD store*.

Tabela 8.1: Abordagens de conversão de RDB para NoSQL, parâmetros de entrada e esquema NoSQL.

Autores	**Entrada**	**Saída**
(Stanescu et al., 2016)	Metadados do RDB	S_a
(Zhao et al., 2014)	Diagrama E-R	S_b
(Karnitis e Arnicans, 2015)	E-R + Classificação de Tabelas + Tabela em Foco: *- Orders, Customers, Products*	S_c
(Jia et al., 2016)	E-R + Classificação de Tabelas/Relacionamentos: *- Frequent Join: Customers/Orders* *- Frequent Join: Orders/Orderlines* *- Frequent Join: Products/Inventory*	S_d

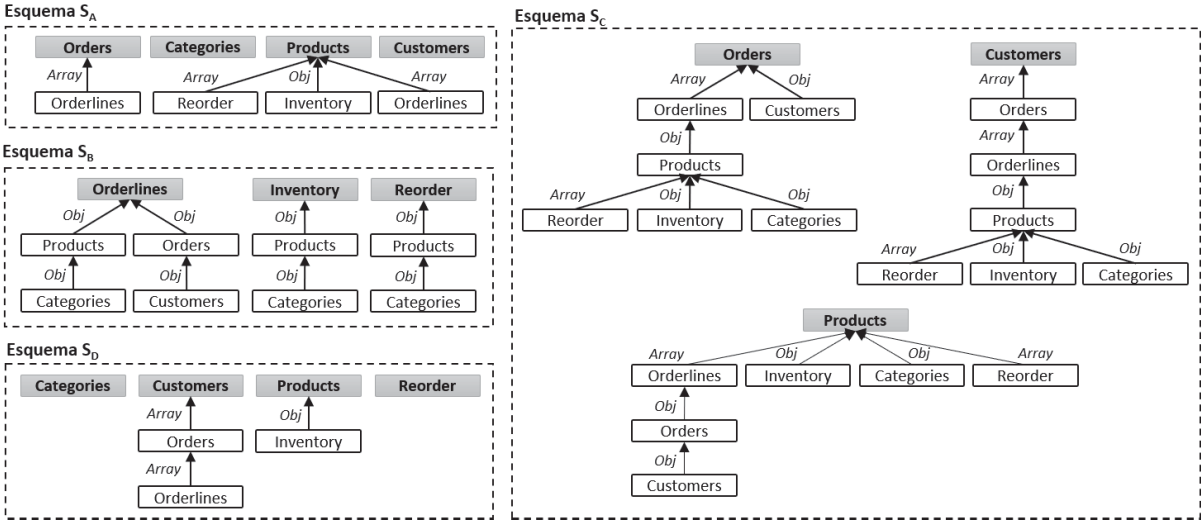


Figura 8.2: Esquemas NoSQL gerados, a partir de regras extraídas de abordagens de conversão de RDB para NoSQL orientado a documentos, provenientes do estado da arte.

Na Figura 8.2, os vértices com cor de fundo em cinza representam as coleções do esquema (vértice raiz). Como pode ser visto, foram gerados esquemas com diferentes números de coleções e diferentes formas de aninhamentos entre entidades. Como todos os esquemas estão representados por meio do mesmo formato é possível compará-los e avaliá-los de forma objetiva.

8.1.2 Definindo o Cenário de Avaliação

O cenário de avaliação considera a melhor correspondência entre o padrão de acesso das consultas e estrutura do esquema. Foram usadas as métricas *Path*, *SubPath*, *IndPath*, *DirEdge*, *AllEdge* e *ReqColls* para verificar se as entidades estão (ou não) aninhadas de acordo com o padrão de acesso. Para calcular o $SScore$ e $QScore$ foram atribuídos os mesmos pesos para todas as consultas e para todas as métricas de caminho ($w_p = 1; w_{sp} = 1; w_{ip} = 1$). Isso significa que todas as consultas e tipos de cobertura de caminhos têm mesma prioridade. A profundidade onde o caminho da consulta é localizado no esquema é considerado no cálculo da métrica, priorizando esquemas na qual as entidades requeridas estão mais perto da raiz do DAG. No entanto, o usuário pode definir diferentes pesos e desconsiderar a profundidade do caminho no cálculo das métricas.

O quadro abaixo apresenta o conjunto de consultas Q. Essas consultas foram selecionadas em função dos diferentes padrões de acesso que elas possuem e por exercitarem as métricas.

```
1  /* Consulta q1 */
2  select * from customers where id_customer = 1;
3  /* Consulta q2 */
4  select * from products inner join inventory on products.id_prod = inventory.prod_id where id_prod = 1;
5  /* Consulta q3 */
6  select * from orders left join orderlines on orderlines.orderid = orders.id_order where id_order = 1;
7  /* Consulta q4 */
8  select * from customers left join orders on customers.id_customer = orders.customerid left join orderlines
9  on orders.id_order = orderlines.orderid left join products on orderlines.prod_id = products.id_prod
10 where orderdate between 2009-01-01 and 2009-01-02;
11 /* Consulta q5 */
12 select * from products left join orderlines on products.id_prod = orderlines.prod_id left join orders
13 on orderlines.orderid = orders.id_order left join customers on orders.customerid = customers.id_customer
14 where products.price between 29 and 30;
15 /* Consulta q6 */
16 select * from orders o left join customers c on o.customerid = c.id_customer left join orderlines ol
17 on ol.orderid = o.id_order where orderdate between 2009-01-01 and 2009-01-02;
18 /* Consulta q7 */
19 select * from inventory right join orderlines on inventory.prod_id = orderlines.prod_id where orderid = 1;
```

Na Figura 8.3 as consultas são exibidas como DAGs, sendo produzidos de acordo com as regras de tradução da seção 4.1.2. Além da estratégia de conversão de SQL para DAG, o usuário pode definir DAGs alternativos para representar diferentes padrões de acesso.

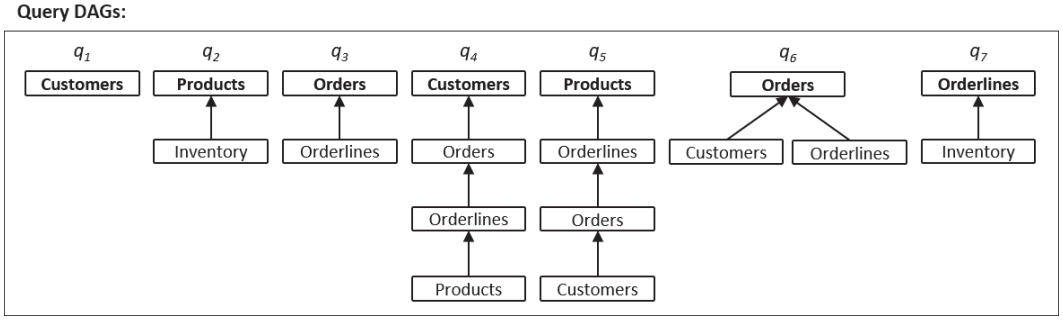


Figura 8.3: Conjunto de consultas do experimento representadas como DAGs.

8.1.3 Resultados do Experimento

Os resultados são apresentados nas seções a seguir, incluindo os valores calculados para as métricas sobre os quatro esquemas, considerações sobre o esforço de implementação das consultas para os esquemas e resultados preliminares sobre o tempo de execução das consultas. A Tabela 8.2 resume todos os resultados e é usada nas seções a seguir.

Tabela 8.2: Resultados obtidos do processo de cálculo de métricas, escores e implementação das consultas sobre os esquemas S_a a S_d. $Paths$ é calculado considerando as métricas $Path$, $SubPath$ e $IndPath$ (ver seção 7.3.1). LoC, $Stages$ e $Time$ são provenientes do processo de implementação e execução das consultas sobre os esquemas.

Esquema S_a							
Consulta	Paths	DirEdge	AllEdge	ReqColls	LoC	Stages	Time
q_1	1.0	0.0	0.0	1	5	1	0.0
q_2	1.0	1.0	1.0	1	11	2	0.0
q_3	1.0	1.0	1.0	1	5	1	0.0
q_4	0.0	0.3	0.3	3	92	11	0.06
q_5	0.0	0.3	0.3	3	73	11	0.89
q_6	0.5	0.5	0.5	2	19	6	0.02
q_7	0.0	0.0	0.0	1	27	5	0.06
SScore	0.50	0.45	0.45	0.58	232	37	1.03
Esquema S_b							
Consulta	Paths	DirEdge	AllEdge	ReqColls	LoC	Stages	Time
q_1	0.3	0.0	0.0	1	16	3	0.06
q_2	0.0	0.0	1.0	1	21	2	0.02
q_3	0.0	0.0	1.0	1	48	2	0.05
q_4	0.0	0.3	1.0	1	66	7	0.07
q_5	0.0	0.66	1.0	1	50	5	0.19
q_6	0.25	0.5	1.0	1	41	6	0.11
q_7	0.0	0.0	0.0	2	23	4	0.06
SScore	0.08	0.21	0.71	0.88	265	29	0.55
Esquema S_c							
Consulta	Paths	DirEdge	AllEdge	ReqColls	LoC	Stages	Time
q_1	1.0	0.0	0.0	1	10	2	0.0
q_2	1.0	1.0	1.0	1	12	2	0.0
q_3	1.0	1.0	1.0	1	11	2	0.0
q_4	1.0	1.0	1.0	1	15	2	0.03
q_5	1.0	1.0	1.0	1	15	2	0.06
q_6	1.0	1.0	1.0	1	13	2	0.01
q_7	0.5	0.0	0.0	1	27	5	0.06
SScore	0.93	0.71	0.71	1.0	103	17	0.16
Esquema S_d							
Consulta	Paths	DirEdge	AllEdge	ReqColls	LoC	Stages	Time
q_1	1.0	0.0	0.0	1	10	2	0.0
q_2	1.0	1.0	1.0	1	12	2	0.0
q_3	0.5	1.0	1.0	1	13	3	0.02
q_4	0.0	0.67	0.66	2	55	9	0.08
q_5	0.0	0.0	0.66	2	89	12	35.91
q_6	0.25	0.5	1.0	1	25	5	0.03
q_7	0.0	0.0	0.0	2	37	8	0.09
SScore	0.39	0.45	0.62	0.70	241	41	36.12

8.1.3.1 Métricas

A Tabela 8.2 exibe o $QScore$ dos esquemas $S_a - S_d$, para as métricas $Paths$, $DirEdge$, $AllEdge$ e $ReqColls$, por consulta. $Paths$ é calculado considerando as métricas $Path$, $SubPath$ e $IndPath$ (ver seção 7.3.1). O $SScore$ de cada esquema também é exibido. Considerando a cobertura de caminhos ($Paths$), o esquema S_c tem o maior escore, sendo $SScore = 0.93$. Isso significa que S_c melhor corresponde ao padrão de acesso do conjunto Q. Na sequência estão os esquemas S_a, S_d e S_b, em que o esquema S_b tem o menor $SScore = 0.08$.

As métricas $Paths$ são usadas para identificar qual esquema melhor cobre as consultas. Por exemplo, no esquema S_c, as consultas $q_1 - q_6$ estão 100% cobertas pelo esquema, por meio das métricas $Path$, $SubPath$ ou $IndPath$. A consulta q_7 é penalizada por estar localizada no nível 2 da coleção $Orders$, resultando em um $QScore$ menor (0.5). Em contraste com o esquema S_b, apenas as consultas q_1 e q_6 têm cobertura via $Paths$, mas ambas consultas são penalizadas devido ao nível onde estão localizadas no esquema. Por exemplo, para responder q_1 é necessário percorrer a coleção $Orderlines$ para encontrar a entidade $Customers$, localizada no nível 3. Ao analisar o esquema S_b, é possível observar que as coleções $Orderlines$, $Inventory$ e $Reorders$ estão invertidas em relação ao padrão de acesso das consultas de Q. Como resultado, não há cobertura $Paths$ para as consultas $q_2 - q_5$ e q_7.

As métricas $DirEdge$ e $AllEdge$ são usadas para verificar se as entidades estão relacionadas umas com as outras no esquema, considerando o padrão de acesso das consultas de Q. Por exemplo, o esquema S_c tem os maiores escores para $DirEdge$ e $AllEdge$. Isso significa que S_c tem o padrão de acesso mais próximo de Q. Em contraste, o esquema S_b tem o maior $AllEdge$ escore (o mesmo valor do esquema S_c) e o menor $DirEdge$ escore, o que significa que as entidades estão relacionadas umas com as outras conforme a estrutura da consulta (DAG), mas a direção do relacionamento está invertida, não correspondendo apropriadamente ao padrão de acesso das consultas. Por exemplo, na coleção $Orderlines$ há correspondência $DirEdge$ parcial para as consultas $q_4 - q_6$, mas para o restante das consultas o esquema S_b está invertido.

Para a métrica $ReqColls$, o esquema S_c tem o melhor resultado, com $SScore = 1.0$. Isso significa que todas as consultas são respondidas acessando os dados de uma única coleção. Os esquemas são ranqueados da seguinte forma: S_c, S_b, S_d e S_a. O esquema S_c tem maior redundância de dados, composto de três coleções que encapsulam todas as entidades do RDB, usando diferentes ordens de aninhamento. No entanto, as entidades de S_c apresentam número maior de elementos do que as entidades dos demais esquemas, o que deve ser considerado ao migrar os dados de RDB para NoSQL.

8.1.3.2 Esforço de Implementação da Consulta

O impacto sobre a implementação da consulta para cada esquema foi medido para avaliar se está relacionado com os resultados das métricas. Para medir o esforço de implementação foi utilizada a métrica de contagem de linhas de código (LoC, do inglês *Lines of Codes*) para implementar a consulta manualmente. Apesar de ser possível gerar as consultas automaticamente, a métrica LoC é interessante para estimar o custo de manutenção das consultas durante o ciclo de vida da aplicação. O objetivo aqui é verificar se esquemas com $SScore$ maior apresentam menor complexidade de implementação das consultas.

Para tal, foram criados quatro bancos de dados no MongoDB, de acordo com os esquemas S_a, S_b, S_c e S_d. MongoDB foi selecionado por ser um dos bancos de dados orientados a documentos amplamente usado pela comunidade de desenvolvedores. Na sequência, foram implementadas todas as consultas da Figura 8.3 usando o *framework Aggregation Pipeline* do MongoDB. Esse *framework* usa o conceito de *pipelines* de processamento de documentos, na

qual cada *pipeline* é composto por um conjunto de estágios. Um estágio recebe um ou mais documentos, executa operações de transformação de dados e retorna os resultados para o próximo estágio, de forma sucessiva até alcançar o último estágio. A implementação das consultas usou um subconjunto das diretrizes definidas no capítulo anterior: *i)* iniciar a implementação na coleção de documentos onde o filtro da consulta está localizado e *ii)* formatar os dados retornados conforme DAG da consulta, executando operações de formatação no sentido vértice folha para vértice raiz do DAG.

O LoC para cada consulta foi obtido por meio do comando $explain$ do MongoDB, que retorna a implementação da consulta com formatação padronizada, facilitando a contagem das linhas. Além do LoC foi computado o número de estágios usados em cada *pipeline* para recuperar e formatar os documentos de acordo com a estrutura do DAG da consulta. A Tabela 8.2 mostra o LoC de cada consulta por esquema. Considerando o LoC total por esquema, S_c tem o menor valor ($LoC = 103$), seguido pelos esquemas S_a, S_d e S_b. Em relação ao número de estágios, S_c tem o menor valor, seguido dos esquemas S_b, S_a e S_d. Neste caso, S_b assume o segundo lugar pelo fato de não usar *arrays* de documentos embutidos em sua estrutura, não necessitando de estágios adicionais para desaninhar (operador *$unwind*) *arrays* de documentos embutidos.

Analisando os resultados de $SScore$ para as métricas $Paths$, $DirEdge$ e $AllEdge$, em conjunto com a soma de LoC e $Stages$, é possível verificar que esquemas com maiores escores para $Paths$ e $DirEdge$ requerem menor LoC para implementar as consultas. Para a métrica $AllEdge$ isso não é sempre verdade. Essa métrica verifica se as entidades do esquema estão relacionadas conforme estrutura do DAG da consulta. No entanto, o relacionamento pode existir, mas a direção pode não ser correspondente ao padrão de acesso da consulta (caso do esquema S_b). Neste caso, mais esforço é necessário para formatar os dados de acordo com o DAG da consulta.

Para concluir, o usuário especialista pode avaliar e comparar as opções de esquema antes de executar a transformação de RDB para NoSQL documento, aplicando o conjunto de métricas e escores definidos neste capítulo. Por meio das métricas é possível verificar se as entidades estão (ou não) aninhadas de acordo com o padrão de acesso da consulta e se é necessário buscar dados de diferentes coleções. Por fim, os resultados obtidos mostram que há relação entre o esforço de implementação da consulta e escores calculados através das métricas.

8.1.3.3 Tempo de Execução das Consultas

O tempo de execução das consultas foi medido para verificar a relação com os resultados das métricas. No entanto, em uma utilização normal da abordagem esta fase não seria necessária, visto que o objetivo é avaliar esquemas antes da migração dos dados. A coluna *Time* da Tabela 8.2 mostra o tempo médio de execução da consulta em segundos, sendo que cada consulta foi executada 30 vezes. É importante notar que o tempo de execução das consultas $q_1 - q_3$ retorna zero segundo para alguns esquemas, em função do filtro da consulta corresponder ao campo de índice na coleção de documentos, retornando os documentos desejados sem ter que percorrer todos os elementos da coleção.

Os resultados mostram que o esquema S_c tem o menor tempo de execução, seguido dos esquemas S_b, S_a e S_d. Esquema S_b ocupa o segundo lugar, apesar de os esquemas S_a e S_d corresponderem melhor ao padrão de acesso das consultas. A razão é devido ao tempo de execução de q_5 para S_a e S_d. Para ambos os esquemas é necessário executar a operação *$lookup* do MongoDB (similar ao comando SQL *left join*). Em S_a é executada a operação *$lookup* entre as coleções $Orders$, $Products$ e $Customers$, enquanto que em S_d entre $Products$ e $Customers$. Ademais, os campos usados na operação *$lookup* estão localizados em *arrays* de documentos embutidos, impactando no tempo de execução. No entanto, esses resultados são preliminares e precisam de investigação futura.

8.2 EXPERIMENTO II: AVALIANDO O IMPACTO DA LOCALIZAÇÃO DO FILTRO DA CONSULTA NO ESQUEMA

O experimento apresentado nesta seção tem por objetivo verificar o impacto da localização do filtro da consulta no esquema, em relação ao esforço de implementação e tempo de execução das consultas, além de demonstrar a aplicação das diretrizes de implementação de consultas definidas na seção 7.5. De forma diferente do experimento anterior, os esquemas e consultas são definidos com foco na avaliação da localização do filtro da consulta. Os resultados serão confrontados com as métricas a fim de verificar sua relação.

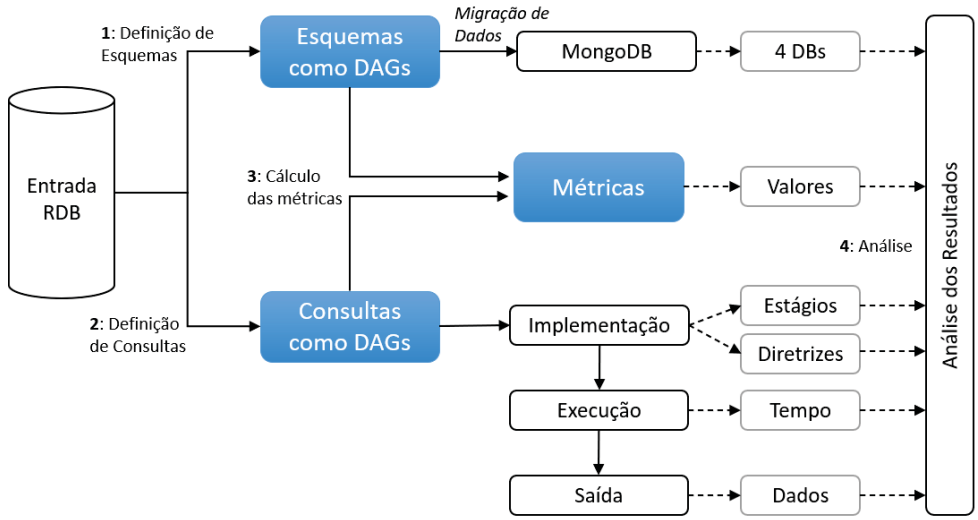


Figura 8.4: Visão geral das etapas que compõem o experimento.

A Figura 8.4 apresenta a visão geral do experimento, que é composto pelas seguintes etapas:

1. **Definição de esquemas**: a partir do RDB de entrada serão criados quatro esquemas com diferentes formas de modelagem das entidades como coleções de documentos. Os esquemas serão representados como DAGs, seguindo a abordagem definida nesta tese. Esses esquemas serão usados para migrar os dados do RDB para o MongoDB, por meio do *framework* Metamorfose. Como resultado serão criados quatro bancos de dados com os mesmos dados, mas estruturados de forma diferente.

2. **Definição de consultas**: seis consultas SQL serão utilizadas para avaliar os diferentes esquemas produzidos no passo anterior. Essas consultas serão convertidas para DAGs e, então, implementadas e executadas sobre os bancos de dados MongoDB.

3. **Cálculo das métricas**: as métricas serão calculadas sobre os esquemas e consultas.

4. **Análise dos resultados**: Os resultados das métricas serão analisados em conjunto com os resultados obtidos ao implementar e executar as consultas sobre os bancos de dados MongoDB.

Nas próximas seções serão apresentados maiores detalhes da execução dos experimentos e resultados obtidos.

8.2.1 Definição de Esquemas

O RDB de entrada é o mesmo utilizado no experimento anterior (Figura 8.1). A partir dele serão criados quatro esquemas ($S_1 - S_4$) usando as entidades *Customers*, *Orders* e *Orderlines*, exibidos na Figura 8.5. Essas três entidades foram selecionadas em função de apresentarem relacionamentos com diferentes cardinalidades. Em cada esquema, a coleção é representada pelo vértice raiz do DAG, com cor de fundo cinza. Os demais vértices são entidades aninhadas. As arestas denotam o tipo de aninhamento entre entidades: documento embutido ou *array* de documentos embutidos.

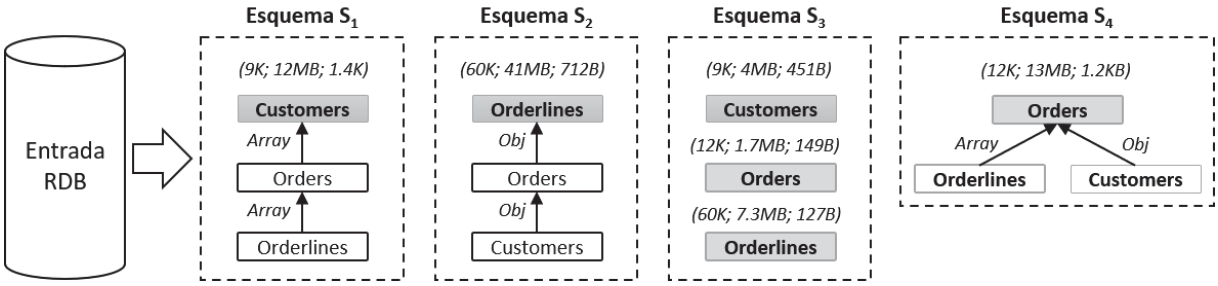


Figura 8.5: Esquemas NoSQL para representar modelagem de documentos. Para cada coleção são exibidos o número de documentos migrados, o tamanho ocupado em disco e o tamanho médio de cada documento, como por exemplo em S_1.*Customers (9K; 12MB; 1.4KB)*.

Para cada esquema foi gerado um banco de dados de mesmo nome no MongoDB. A migração dos dados foi realizada pelo *framework* Metamorfose. Como resultado, todos os bancos de dados têm os mesmos dados, porém estruturados de maneira distinta:

- Esquema S_1: dados aninhados, sem redundância. A direção do aninhamento segue ordem 1-N e são usados *arrays* de documentos embutidos.
- Esquema S_2: dados aninhados, com redundância. A direção do aninhamento segue ordem N-1, sem uso de *arrays* de documentos embutidos. Como resultado, são armazenadas várias cópias das entidades *Orders* e *Customers*.
- Esquema S_3: dados não aninhados, sem redundância. Modelo similar ao relacional, com coleções independentes. Necessário junção de documentos entre coleções para responder as consultas.
- Esquema S_4: dados aninhados, com redundância parcial (várias cópias da entidade *Customers*). Há aninhamentos nas direções 1-N e N-1.

Os quatro bancos de dados representam diferentes formas de relacionar as entidades, possibilitando implementar consultas em que o filtro está localizado em diferentes níveis no esquema. Na Figura 8.5, além das coleções também são exibidos o número de documentos migrados, o tamanho ocupado em disco e o tamanho médio de cada documento. Considerando a coleção S_1.*Customers (9K; 12MB; 1.4KB)*, há 9K documentos armazenados, a coleção ocupa 12MB em disco e o tamanho médio dos documentos é de 1.4KB. O maior esquema em termos de espaço em disco é S_2, com 60K documentos, 41MB de espaço em disco e 712B de tamanho médio por documento. O tamanho de cada esquema reflete como os dados foram estruturados.

8.2.2 Definição de Consultas

Seis consultas foram definidas para avaliar os esquemas apresentados anteriormente (Figura 8.6). Dois grupos de consultas foram definidos, de acordo com a estrutura do DAG. As consultas $q_1 - q_3$ pertencem ao *Grupo 1* e as consultas $q_4 - q_6$ pertencem ao *Grupo 2*. Em cada grupo há uma instrução SQL principal, um conjunto de filtros (predicados) e respectivo DAG. A combinação entre instrução principal e filtro define a consulta. No *Grupo 1* há uma instrução principal e três filtros para definir as consultas q_1, q_2 e q_3. O filtro de cada consulta está localizado em uma entidade diferente (representado pela letra *F* na Figura 8.6). A mesma abordagem é usada para definir o *Grupo 2*. Esse arranjo de consultas e filtros permite verificar se a localização do filtro tem impacto na implementação e execução das consultas.

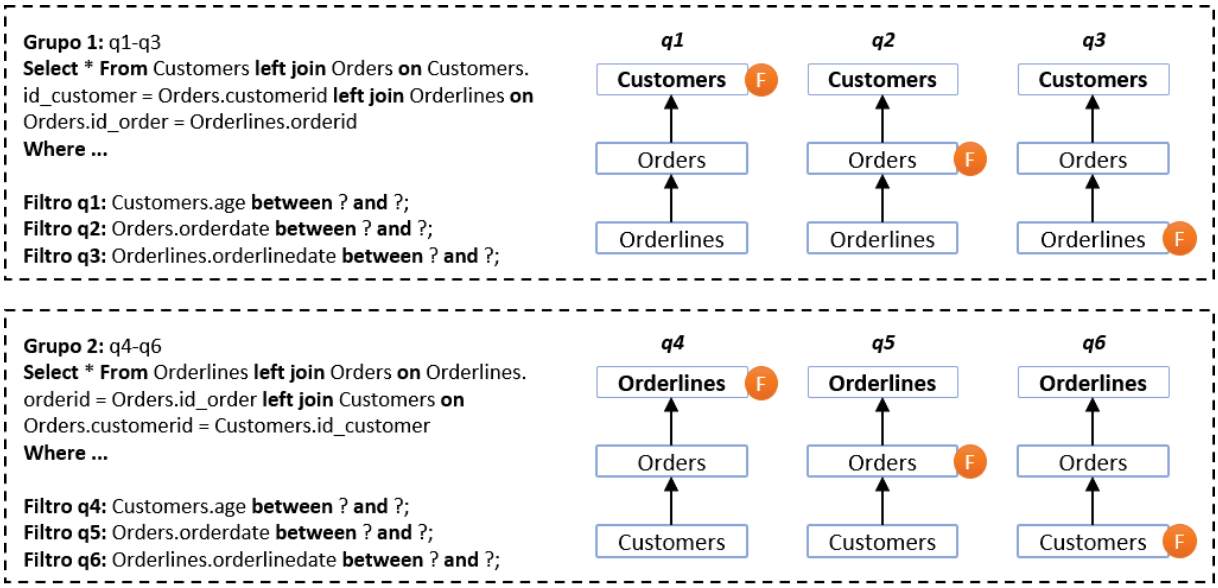


Figura 8.6: Consultas usadas no experimento representadas como SQL e DAGs.

O conjunto de consultas foi implementado para cada um dos esquemas, seguindo as diretrizes definidas na seção 7.5. No total foram implementadas 24 consultas por meio do *framework Aggregation Pipeline*. Os dados retornados seguem a estrutura do DAG da consulta.

8.2.3 Resultados

Nesta seção serão apresentados os resultados dos cálculos das métricas, da implementação e da execução das consultas q_1 a q_6, sobre os esquemas $S_1 - S_4$. Para determinar qual esquema fornece maior cobertura para uma consulta, os resultados das métricas são interpretados considerando a seguinte ordem: menor valor para $ReqColls$ e maior valor para $Path$, $SubPath$, $IndPath$, $DirEdge$ e $AllEdge$. Ademais, os dados retornados ao executar cada uma das consultas sobre cada um dos esquemas foram verificados e são consistentes entre si.

8.2.3.1 Consultas q_1 a q_3

A Tabela 8.3 exibe os valores das métricas para as consultas q_1 a q_3. É importante notar que é exibido apenas um resultado por esquema, pois os DAGs das consultas têm mesma estrutura e, consequentemente, os mesmos valores para as métricas. De acordo com os valores exibidos, o esquema S_1 apresenta maior correspondência estrutural para as três consultas. Na

sequência estão os esquemas S_4, S_2 e S_3. Para S_3 é necessário realizar a junção de três coleções para responder as consultas.

Tabela 8.3: Resultados das métricas de cobertura para as consultas q_1, q_2 e q_3, por esquema.

Esquema	Path	SubPath	IndPath	DirEdge	AllEdge	ReqColls
S_1	1.0	1.0	0.0	1.0	1.0	1
S_2	0.0	0.0	0.0	0.0	1.0	1
S_3	0.0	0.0	0.0	0.0	0.0	3
S_4	0.0	0.0	0.0	0.5	1.0	1

A Tabela 8.4 apresenta o conjunto de operadores e a ordem na qual eles são utilizados na implementação das consultas, usando o *framework MongoDB Aggregation Pipeline*. As colunas da tabela mostram o nome do esquema e a coleção de início da implementação do *pipeline*.

Tabela 8.4: Operadores usados para implementar $q_1 - q_3$, sobre os esquemas $S_1 - S_4$.

	$S_1.Customers$		$S_2.Orderlines$		$S_3.Customers$		$S_4.Orders$
	1 Match(Customers)	1	Match(Customers)	1	Match(Customers)	1	Match(Customers)
		2	Project	2	Lookup(Orders)	2	Group(id_customer)
		3	Project	3	Unwind(Orders)	3	ReplaceRoot
q_1		4	Group(id_order)	4	Lookup(Orderlines)	4	Project
		5	Project	5	Group(id_customer)		
		6	Group(id_customer)	6	Project		
		7	ReplaceRoot	7	ReplaceRoot		
	$S_1.Customers$		$S_2.Orderlines$		$S_3.Orders$		$S_4.Orders$
	1 Match(Orders)	1	Match(Orders)	1	Match(Orders)	1	Match(Orders)
	2 Unwind(Orders)	2	Project	2	Lookup(Orderlines)	2	Group(id_customer)
	3 Match(Orders)	3	Project	3	Group(id_customer)	3	ReplaceRoot
q_2	4 Group(id_customer)	4	Group(id_order)	4	Lookup(Customers)	4	Project
	5 AddField	5	Project	5	Unwind(Customers)		
	6 ReplaceRoot	6	Group(id_customer)	6	ReplaceRoot		
		7	ReplaceRoot				
	$S_1.Customers$		$S_2.Orderlines$		$S_3.Orderlines$		$S_4.Orders$
	1 Match(Orderlines)	1	Match(Orderlines)	1	Match(Orderlines)	1	Match(Orderlines)
	2 Unwind(Orders)	2	Project	2	Lookup(Orders)	2	Unwind(Orderlines)
	3 Unwind(Orderlines)	3	Project	3	Unwind(Orders)	3	Match(Orderlines)
	4 Match(Orderlines)	4	Group(id_order)	4	Group(id_order)	4	Group(id_order)
q_3	5 Group(id_order)	5	Project	5	AddField	5	AddField
	6 AddField	6	Group(id_customer)	6	Lookup(Customers)	6	Group(id_customer)
	7 Group(id_customer)	7	ReplaceRoot	7	Unwind(Customers)	7	ReplaceRoot
	8 AddField			8	Group(id_customer)	8	Project
	9 ReplaceRoot			9	ReplaceRoot		

A implementação das consultas $q_1 - q_3$ sobre o esquema S_1 usa um, seis e nove operadores, respectivamente. Na implementação de q_1 o filtro da consulta está localizado no $level = 1$ da coleção $S_1.Customers$ e não há necessidade de formatação dos documentos, pois a estrutura da consulta corresponde à estrutura do esquema. No entanto, para as consultas q_2 e q_3 há custo de formatação. Esse custo é devido a localização do filtro em campo de *array* de documentos embutidos. É necessário buscar os documentos, desagrupar, aplicar operador $\$match$ (operador 3 em q_2 e operador 4 em q_3) e reagrupar os documentos conforme DAG das consultas. Essas operações impactam a implementação das consultas q_2 e q_3, independente da correspondência estrutural entre os DAGs das consultas e esquema S_1.

As consultas $q_1 - q_3$ estão invertidas em relação ao esquema S_2. A implementação das três consultas é semelhante, com sete operadores e alterações somente no operador $\$match$, para refletir o filtro da consulta. Em comparação a S_1, o esquema S_2 não usa *arrays* para relacionar os dados, não sendo necessária operações extras. No entanto, há custo de formatação em função do esquema não corresponder aos DAGs das consultas.

Para S_3, as consultas necessitam de junção de documentos de diferentes coleções. A implementação (operadores) de cada uma delas depende da localização do filtro no esquema, conforme diretrizes apresentadas anteriormente. Para q_1, a coleção de início é $Customers$ e são necessários sete operadores. Para q_2, a coleção de início é $Orders$ e são necessários seis operadores. Para q_3 a coleção de início é $Orderlines$ e são necessários nove operadores.

A implementação das consultas $q_1 - q_3$ sobre o esquema S_4 usa quatro, quatro e oito operadores, respectivamente. As consultas q_1 e q_2 têm implementações semelhantes, alterando apenas o operador $\$match$, para refletir o filtro da consulta. No entanto, para q_3 o filtro está localizado em *array* de documentos embutidos. Neste caso, é necessário buscar, desagrupar, re-filtrar e reagrupar os documentos conforme DAG da consulta.

A Tabela 8.5 apresenta o tempo de execução das consultas para os esquemas $S_1 - S_4$, variando a seletividade da consulta. A seletividade é definida como a porcentagem de documentos retornados do banco de dados ($Sl_x\%$). Df representa o número final de documentos retornados, após a execução de todos os operadores da consulta. Tf representa o tempo em segundos transcorrido para buscar os documentos no banco de dados e formatá-los conforme o DAG da consulta.

Tabela 8.5: Tempo de execução e número de documentos retornados para as consultas $q_1 - q_3$.

Consulta q_1												
Esquema	$Sl_1(\%)$	Df	Tf	$Sl_2(\%)$	Df	Tf	$Sl_3(\%)$	Df	Tf	$Sl_4(\%)$	Df	Tf
S_1	7%	600	0.01	25%	2221	0.01	52%	4671	0.01	100%	8996	0.01
S_2	7%	600	0.18	25%	2221	0.42	52%	4671	0.77	100%	8996	1.47
S_3	7%	600	81.32	25%	2221	279.96	52%	4671	564.02	100%	8996	1065.12
S_4	7%	600	0.05	25%	2221	0.13	52%	4671	0.27	100%	8996	0.47
Consulta q_2												
Esquema	$Sl_1(\%)$	Df	Tf	$Sl_2(\%)$	Df	Tf	$Sl_3(\%)$	Df	Tf	$Sl_4(\%)$	Df	Tf
S_1	0%	32	0.02	22%	1972	0.09	58%	5212	0.25	100%	8996	0.44
S_2	0%	32	0.07	17%	1972	0.29	50%	5212	0.72	100%	8996	1.43
S_3	0%	32	3.60	17%	1972	194.89	50%	5212	566.64	100%	8996	1130.95
S_4	0%	32	0.02	17%	1972	0.10	50%	5212	0.27	100%	8996	0.47
Consulta q_3												
Esquema	$Sl_1(\%)$	Df	Tf	$Sl_2(\%)$	Df	Tf	$Sl_3(\%)$	Df	Tf	$Sl_4(\%)$	Df	Tf
S_1	0%	32	0.04	22%	1972	0.19	58%	5212	0.41	100%	8996	0.68
S_2	0%	32	0.05	17%	1972	0.28	50%	5212	0.72	100%	8996	1.46
S_3	0%	32	3.07	17%	1972	193.71	50%	5212	552.67	100%	8996	1098.03
S_4	0%	32	0.03	17%	1972	0.16	50%	5212	0.36	100%	8996	0.73

Apesar das consultas $q_1 - q_3$ compartilharem o mesmo DAG, o número de operadores e tempo de execução difere em função da localização do filtro da consulta no esquema. A consulta q_1 tem o menor Tf quando executada sobre o esquema S_1. Inclusive, o Tf é constante para todas as opções de seletividade ($Sl_1 - Sl_4$), pois não há necessidade de formatação dos dados devido a correspondência estrutural e o filtro da consulta estar localizado na entidade raiz (nível 1) da coleção. De forma diferente, os esquemas S_4 e S_2 apresentam custo de formatação para q_1, que aumenta conforme aumenta o número de documentos retornados pela consulta ($Sl_x\%$).

A execução das consultas q_2 e q_3 sobre S_1 é impactada negativamente, devido a localização do filtro em *array* de documentos embutidos. A consulta q_3 sobre S_4 tem Tf ligeiramente inferior em relação a S_1. No entanto, quando q_3 retorna 100% dos documentos da coleção o Tf de S_1 é menor que o Tf de S_4 em função da diferença do número de documentos entre $S_1.Customers$ e $S_4.Orders$, em que esta última tem número maior.

A execução das consultas sobre S_3 envolve a junção de documentos de diferentes coleções. O custo para buscar, juntar, agrupar e formatar os documentos é superior do que nos demais esquemas. Inclusive, esse custo aumenta conforme a porcentagem (Sl_x) de documentos retornados pela consulta aumenta. **Nota**: o tempo de execução das consultas sobre S_3 pode ser melhorado com o uso de índices nas coleções de documentos para otimizar as operações de junção. No entanto, o objetivo do experimento não é otimizar a execução das consultas.

8.2.3.2 Consultas q_4 a q_6

A Tabela 8.6 exibe os valores das métricas para as consultas q_4 a q_6. De acordo com os valores exibidos, o esquema S_2 apresenta maior correspondência estrutural para as três consultas. Na sequência estão os esquemas S_4, S_1 e S_3. Para S_3 é necessário realizar a junção de três coleções para responder as consultas.

Tabela 8.6: Resultados das métricas de cobertura para as consultas q_4, q_5 e q_6, por esquema.

Esquema	Path	SubPath	IndPath	DirEdge	AllEdge	ReqColls
S_1	0.0	0.0	0.0	0.0	1.0	1
S_2	1.0	1.0	0.0	1.0	1.0	1
S_3	0.0	0.0	0.0	0.0	0.0	3
S_4	0.0	0.0	0.0	0.5	1.0	1

A Tabela 8.7 apresenta os operadores utilizados na implementação das consultas $q_4 - q_6$ para os esquemas S_1 a S_4. As consultas $q_4 - q_6$ estão invertidas em relação a $S_1.Customers$. Para q_4, são necessários oito operadores, enquanto que para q_5 e q_6 são necessários nove operadores, para buscar, desagrupar e formatar os documentos conforme DAG das consultas. Além disso, o filtro de q_5 e q_6 está localizado em *array* de documentos embutidos, sendo necessário adicionar um operador *$match* extra (operador 4). Esse operador é adicionado para garantir que somente os documentos relacionados ao filtro serão retornados ao usuário.

De forma diferente, para S_2 as consultas correspondem a estrutura do esquema (DAG). A implementação de q_4, q_5 e q_6 usa apenas o operador *$match*. Além disso, como $S_2.Orderlines$ não relaciona as entidades por meio de *arrays* de documentos embutidos, não há impacto ao variar a localização do filtro da consulta.

Em relação ao esquema S_3 são necessárias operações de junção de documentos de diferentes coleções. A implementação de cada uma delas depende da localização do filtro no esquema. Para q_4, a coleção de início é $Customers$ e são necessários nove operadores. Para q_5, a coleção de início é $Orders$ e são necessários oito operadores. Por fim, para q_6 a coleção de início é $Orderlines$ e são necessários cinco operadores.

As consultas $q_4 - q_6$ correspondem parcialmente ao DAG de $S_4.Orders$. A implementação de q_4 e q_5 é semelhante, consistindo no uso de quatro operadores. Para q_6 é adicionado um operador *$match* adicional (operador 3), devido ao filtro da consulta estar localizado em *array* de documentos embutidos ($S_4.Orders.Orderlines$).

A Tabela 8.8 apresenta o tempo de execução das consultas $q_4 - q_6$ para os esquemas $S_1 - S_4$, variando a seletividade da consulta. O esquema S_2 tem o menor Tf para todas as consultas,

Tabela 8.7: Operadores usados para implementar $q_4 - q_6$, sobre os esquemas $S_1 - S_4$.

	$S_1.Customers$	$S_2.Orderlines$	$S_3.Customers$	$S_4.Orders$
	1 Match(Customers)	1 Match(Customers)	1 Match(Customers)	1 Match(Customers)
	2 Unwind(Orders)		2 Lookup(Orders)	2 Unwind(Orderlines)
	3 Unwind(Orderlines)		3 Unwind(Orders)	3 ReplaceRoot
	4 Project		4 Lookup(Orderlines)	4 Project
q_4	5 AddFields		5 Unwind(Orderlines)	
	6 AddFields		6 AddFields	
	7 Project		7 AddFields	
	8 ReplaceRoot		8 Project	
			9 ReplaceRoot	
	$S_1.Customers$	$S_2.Orderlines$	$S_3.Orders$	$S_4.Orders$
	1 Match(Orders)	1 Match(Orders)	1 Match(Orders)	1 Match(Orders)
	2 Unwind(Orders)		2 Lookup(Orderlines)	2 Unwind(Orderlines)
	3 Unwind(Orderlines)		3 Unwind(Orderlines)	3 ReplaceRoot
	4 Match(Orders)		4 AddFields	4 Project
q_5	5 Project		5 Lookup(Customers)	
	6 AddField		6 Unwind(Customers)	
	7 AddField		7 ReplaceRoot	
	8 Project		8 Project	
	9 ReplaceRoot			
	$S_1.Customers$	$S_2.Orderlines$	$S_3.Orderlines$	$S_4.Orders$
	1 Match(Orderlines)	1 Match(Orderlines)	1 Match(Orderlines)	1 Match(Orderlines)
	2 Unwind(Orders)		2 Lookup(Orders)	2 Unwind(Orderlines)
	3 Unwind(Orderlines)		3 Unwind(Orders)	3 Match(Orderlines)
	4 Match(Orderlines)		4 Lookup(Customers)	4 ReplaceRoot
q_6	5 Project		5 Unwind(Customers)	5 Project
	6 AddField			
	7 AddField			
	8 Project			
	9 ReplaceRoot			

Tabela 8.8: Tempo de execução e número de documentos retornados para as consultas $q_4 - q_6$.

Consulta q_4												
Esquema	Sl_1(%)	Df	Tf	Sl_2(%)	Df	Tf	Sl_3(%)	Df	Tf	Sl_4(%)	Df	Tf
S_1	7%	4047	0.14	25%	15132	0.57	52%	31303	1.16	100%	60350	2.22
S_2	7%	4047	0.13	25%	15132	0.11	52%	31303	0.16	100%	60350	0.23
S_3	7%	4047	81.74	25%	15132	290.26	52%	31303	635.21	100%	60350	1193.17
S_4	7%	4047	0.09	25%	15132	0.34	52%	31303	0.71	100%	60350	1.33
Consulta q_5												
Esquema	Sl_1(%)	Df	Tf	Sl_2(%)	Df	Tf	Sl_3(%)	Df	Tf	Sl_4(%)	Df	Tf
S_1	0%	163	0.02	22%	10460	0.47	58%	30453	1.23	100%	60350	2.37
S_2	0%	163	0.11	17%	10460	0.10	50%	30453	0.12	100%	60350	0.21
S_3	0%	163	5.09	17%	10460	319.57	50%	30453	927.01	100%	60350	1806.83
S_4	0%	163	0.02	17%	10460	0.29	50%	30453	0.74	100%	60350	1.30
Consulta q_6												
Esquema	Sl_1(%)	Df	Tf	Sl_2(%)	Df	Tf	Sl_3(%)	Df	Tf	Sl_4(%)	Df	Tf
S_1	0%	163	0.07	22%	10460	0.55	58%	30453	1.33	100%	60350	2.54
S_2	0%	163	0.08	17%	10460	0.12	50%	30453	0.10	100%	60350	0.10
S_3	0%	163	4.73	17%	10460	300.25	50%	30453	884.79	100%	60350	1780.25
S_4	0%	163	0.05	17%	10460	0.34	50%	30453	0.75	100%	60350	1.43

seguido pelos esquemas S_4, S_1 e S_3. O esquema S_2 não apresenta custo de formatação, devido a correspondência estrutural entre as consultas e a coleção $S_2.Orderlines$. Em contrapartida, o esquema S_1 apresenta alto custo de formatação em função da coleção $S_1.Customers$ estar

invertida em relação aos DAGs das consultas e por usar *arrays* de documentos embutidos para aninhar as entidades. Neste caso, primeiro é necessário desagrupar as entidades de S_1 para depois formatar os dados conforme DAG de $q_4 - q_6$. O custo dessas operações impacta negativamente o Tf do esquema S_1, conforme é visto na Tabela 8.8.

Para S_4, as consultas $q_4 - q_6$ correspondem parcialmente ao DAG da coleção $S_4.Orders$. O custo consiste em aninhar $Orders$ em $Orderlines$ como um documento embutido. O esquema S_3 apresenta maior Tf dentre os esquemas devido as operações de junção (*$lookup*) e formatação de documentos de diferentes coleções.

8.2.4 Análise dos Resultados

A Tabela 8.9 resume os resultados das métricas, número de estágios usados na implementação e tempo de execução das consultas $q_1 - q_6$ sobre os esquemas $S_1 - S_4$.

Neste experimento, todas as consultas têm mesmo peso, assim como as métricas $Path$, $SubPath$ e $IndPath$ (os pesos das métricas são considerados no cálculo de $PATHs$). Em relação aos resultados das métricas, os esquemas S_1 e S_2 apresentam mesmo valor para $PATHs$, $DirEdge$, $AllEdge$ e $ReqCols$, em que as consultas $q_1 - q_3$ correspondem a estrutura de S_1 e as consultas $q_4 - q_6$ correspondem a estrutura de S_2. O esquema S_4 tem correspondência parcial com todas as consultas ($DirEdge = 0.5$). O esquema S_3 não apresenta correspondência estrutural com nenhuma das consultas e necessita de junção de documentos de diferentes coleções para respondê-las ($ReqColls < 1.0$).

Considerando o número de operadores ou estágios do *pipeline* para implementar as consultas, os esquemas podem ser ordenados da seguinte forma: S_2, S_4, S_1 e S_3. O esquema S_2 não usa *arrays* de documentos embutidos em sua estrutura e não é impactado pela localização do filtro da consulta sobre esse tipo de campo. O esquema S_4 sofre impacto em função da localização do filtro das consultas q_3 e q_4, em que ambas estão sobre *arrays* de documentos embutidos, requerendo operadores extras para filtrar os dados corretamente. O esquema S_1 é o mais impactado, em que as implementações das consultas q_2, q_3, q_5 e q_6 necessitam de operadores extras em função da localização do filtro da consulta. Por fim, o esquema S_3 tem número de estágios próximo de S_1 e esse número varia em função de qual coleção é usada como início para implementar o *pipeline* da consulta. Os resultados mostram que a localização do filtro tem impacto na implementação da consulta, e que deve ser considerada no processo de seleção do esquema.

Considerando o tempo total para executar o conjunto de consultas, S_2 tem o menor Tf para as seletividades $Tf(Sl_2)$, $Tf(Sl_3)$ e $Tf(Sl_4)$, seguido pelos esquemas S_4, S_1 e S_3. O esquema S_2 é superado por S_1 e S_4 apenas para $Tf(Sl_1)$, em função da diferença de tamanho (número de documentos) entre as coleções. O tempo de execução total para S_1 é impactado pelas consultas $q_4 - q_6$, que apresentam padrão de acesso invertido em relação ao esquema e seguem ordem de aninhamento $1 - N$. De forma diferente, o esquema S_4 tem correspondência parcial com todas as consultas do experimento ($DirEdge = 0.5$), incluindo as consultas $q_4 - q_6$, o que impacta positivamente seu tempo de execução total. O esquema S_2 está invertido em relação ao padrão de acesso das consultas $q_1 - q_3$. Apesar disso, o impacto de implementação e execução é menor em comparação ao esquema S_1, pelo fato de não usar *arrays* em sua estrutura. Por fim, o esquema S_3 necessita de junção e formatação de documentos de diferentes coleções para responder todas as consultas, o que impactou negativamente o tempo de execução total.

Ao analisar os resultados é possível concluir que o cenário ideal é quando há correspondência estrutural entre esquema e consulta, e a localização do filtro da consulta não incide sobre campo *array* de documentos embutidos. Caso contrário, operadores adicionais são necessários para garantir que somente os dados relacionados ao filtro da consulta serão

retornados, impactando negativamente o esforço de implementação e tempo de execução da consulta. O pior cenário identificado é quando o esquema segue ordem de aninhamento $1-N$ e o padrão de acesso da consulta está invertido a estrutura do esquema. Neste caso, há necessidade de desagrupar os documentos para depois formatá-los conforme DAG da consulta.

Neste experimento as consultas foram configuradas com o mesmo peso, assim como as métricas $Path$, $SubPath$ e $IndPath$. Esse cenário é interessante para verificar a correlação entre as métricas, esforço de implementação e tempo de execução das consultas. No entanto, outros cenários podem ser definidos, em que as consultas podem ter prioridades distintas (consultas executadas frequentemente tem maior prioridade, por exemplo). Ao configurar diferentes pesos para métricas e consultas o ranqueamento dos esquemas é afetado, refletindo diferentes requisitos da aplicação.

Outro fator importante é o tamanho das coleções de documentos do esquema. O número e tamanho dos documentos de uma coleção impactam no tempo de execução das consultas, principalmente quando há várias operações de formatação sobre os documentos. As métricas definidas nesta tese tem por objetivo principal guiar o usuário especialista no processo de seleção do esquema NoSQL na fase de modelagem. Nesta fase, o tamanho e número de instâncias que serão migradas para a base NoSQL não são necessariamente conhecidas. Apesar disso, as métricas fornecem subsídios para identificar quais consultas exigirão maior esforço de implementação e quais terão impacto no tempo de execução em função das características estruturais dos esquemas.

Tabela 8.9: Resultados dos cálculos das métricas, implementação e tempo de execução das consultas, agrupados por esquema.

Esquema S_1	Cobertura			QScore				Implementação e Tempo de Execução				
Consulta	Path	SubPath	IndPath	PATHs	DirEdge	AllEdge	ReqColls	Stages	Tf(Sl_1)	Tf(Sl_2)	Tf(Sl_3)	Tf(Sl_4)
q_1	1.0	1.0	0.0	1.0	1.0	1.0	1	1	0.01	0.01	0.01	0.01
q_2	1.0	1.0	0.0	1.0	1.0	1.0	1	6	0.02	0.09	0.25	0.44
q_3	1.0	1.0	0.0	1.0	1.0	1.0	1	9	0.04	0.19	0.41	0.68
q_4	0.0	0.0	0.0	0.0	0.0	1.0	1	8	0.14	0.57	1.16	2.22
q_5	0.0	0.0	0.0	0.0	0.0	1.0	1	9	0.02	0.47	1.23	2.37
q_6	0.0	0.0	0.0	0.0	0.0	1.0	1	9	0.07	0.55	1.33	2.54
SScore				0.5	0.5	1.0	1.0	42	0.30	1.88	4.39	8.25
Esquema S_2	Cobertura			QScore				Implementação e Tempo de Execução				
Consulta	Path	SubPath	IndPath	PATHs	DirEdge	AllEdge	ReqColls	Stages	Tf(Sl_1)	Tf(Sl_2)	Tf(Sl_3)	Tf(Sl_4)
q_1	0.0	0.0	0.0	0.0	0.0	1.0	1	7	0.18	0.42	0.77	1.47
q_2	0.0	0.0	0.0	0.0	0.0	1.0	1	7	0.07	0.29	0.72	1.43
q_3	0.0	0.0	0.0	0.0	0.0	1.0	1	7	0.05	0.28	0.72	1.46
q_4	1.0	1.0	0.0	1.0	1.0	1.0	1	1	0.13	0.11	0.16	0.23
q_5	1.0	1.0	0.0	1.0	1.0	1.0	1	1	0.11	0.10	0.12	0.21
q_6	1.0	1.0	0.0	1.0	1.0	1.0	1	1	0.08	0.12	0.10	0.10
SScore				0.5	0.5	1.0	1.0	24	0.61	1.31	2.59	4.91
Esquema S_3	Cobertura			QScore				Implementação e Tempo de Execução				
Consulta	Path	SubPath	IndPath	PATHs	DirEdge	AllEdge	ReqColls	Stages	Tf(Sl_1)	Tf(Sl_2)	Tf(Sl_3)	Tf(Sl_4)
q_1	0.0	0.0	0.0	0.0	0.0	0.0	3	7	81.32	279.96	564.02	1065.12
q_2	0.0	0.0	0.0	0.0	0.0	0.0	3	6	3.60	194.89	566.64	1130.95
q_3	0.0	0.0	0.0	0.0	0.0	0.0	3	9	3.07	193.71	552.67	1098.03
q_4	0.0	0.0	0.0	0.0	0.0	0.0	3	9	81.74	290.26	635.21	1193.17
q_5	0.0	0.0	0.0	0.0	0.0	0.0	3	8	5.09	319.57	927.01	1806.83
q_6	0.0	0.0	0.0	0.0	0.0	0.0	3	5	4.73	300.25	884.79	1780.25
SScore				0.0	0.0	0.0	0.3	44	179.55	1578.64	4130.34	8074.35
Esquema S_4	Cobertura			QScore				Implementação e Tempo de Execução				
Consulta	Path	SubPath	IndPath	PATHs	DirEdge	AllEdge	ReqColls	Stages	Tf(Sl_1)	Tf(Sl_2)	Tf(Sl_3)	Tf(Sl_4)
q_1	0.0	0.0	0.0	0.0	0.5	1.0	1	4	0.05	0.13	0.27	0.47
q_2	0.0	0.0	0.0	0.0	0.5	1.0	1	4	0.02	0.10	0.27	0.47
q_3	0.0	0.0	0.0	0.0	0.5	1.0	1	8	0.03	0.16	0.36	0.73
q_4	0.0	0.0	0.0	0.0	0.5	1.0	1	4	0.09	0.34	0.71	1.33
q_5	0.0	0.0	0.0	0.0	0.5	1.0	1	4	0.02	0.29	0.74	1.30
q_6	0.0	0.0	0.0	0.0	0.5	1.0	1	5	0.05	0.34	0.75	1.43
SScore				0.0	0.5	1.0	1.0	29	0.26	1.36	3.09	5.73

8.3 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Este capítulo apresentou dois experimentos em um cenário de conversão de RDB para NoSQL orientado a documentos, em que vários esquemas NoSQL candidatos são avaliados em relação a um conjunto de consultas previamente definido.

O primeiro experimento teve por objetivo validar as métricas considerando quatro esquemas NoSQL gerados a a partir abordagens de conversão do estado da arte e um conjunto de consultas previamente conhecido, que representa o padrão de acesso da aplicação. Os resultados mostraram que as métricas são efetivas para auxiliar o usuário no processo de seleção do esquema adequado ao padrão de acesso da aplicação. No entanto, neste experimento não foi realizada uma análise detalhada do processo de implementação das consultas. Ademais, os resultados deste experimento foram apresentados na conferência CAiSE'20 (Kuszera et al., 2020b).

O segundo experimento investigou fatores que impactam na implementação e execução de consultas sobre esquemas NoSQL. Neste experimento a implementação das consultas seguiu as diretrizes definidas no Capítulo 7, em que são definidas estratégias para implementar as consultas conforme a estrutura do esquema. Os resultados mostraram que o uso de *arrays* de documentos embutidos para aninhar as entidades e a localização do filtro em relação ao esquema impactam na implementação da consulta. Além disso, os resultados mostraram a relação entre as métricas e esforço de implementação das consultas, em que o cenário ideal é quando há correspondência estrutural entre esquema e consulta, e a localização do filtro da consulta não incide sobre campo *array* de documentos embutidos. Quando o filtro da consulta está localizado sobre *array* de documentos embutidos são necessários operadores adicionais para garantir que somente os dados relacionados ao filtro da consulta serão retornados, impactando negativamente na implementação e tempo de execução da consulta. Como trabalho futuro pretende-se incluir a localização do filtro da consulta no DAG e considerar no cálculo das métricas.

9 CONCLUSÕES

Este trabalho de doutorado pesquisou estratégias para converter bancos de dados relacionais para bancos de dados NoSQL. O processo de conversão de bases de dados relacionais para bancos de dados NoSQL não é uma tarefa trivial. Bancos de dados NoSQL fornecem modelos de dados flexíveis, permitindo estruturar os dados de diversas formas. A escolha do formato adequado depende de vários aspectos, como o padrão de acesso da aplicação, o nível de redundância de dados, o tamanho da base de dados, o esforço de manutenção da aplicação, entre outros. O foco desta tese consistiu no padrão de acesso da aplicação e esforço de manutenção da aplicação.

A primeira contribuição desta tese é a definição de uma abordagem de conversão de RDB para NoSQL orientado a documentos e orientado a família de colunas. A abordagem usa grafos acíclicos direcionados (DAGs) como meio de definir a conversão e migração de instâncias do RDB para o modelo NoSQL destino. Um conjunto de DAGs é usado como uma abstração para representar um esquema NoSQL (ou forma de estruturação dos dados). O esquema NoSQL é usado como entrada pelo Metamorfose, um *framework* de transformação de dados baseado no *Apache Spark* definido nesta tese.

O Metamorfose fornece funções para carregar dados em memória, definir mapeamentos entre campos de origem e destino, executar transformações e persistir os dados. Os mapeamentos encapsulam a lógica de transformação de dados por meio de funções definidas pelo usuário (FDU) em Java ou Javascript. Esse mecanismo fornece flexibilidade para estender o *framework* para novos cenários. A transformação de dados é executada por meio de funções *map* e *mapreduce*. Ademais, os mapeamentos são armazenados em formato JSON, permitindo sua reutilização ou auditoria das transformações de dados. A versão atual do Metamorfose suporta arquivos CSV e JSON, e fontes de dados JDBC.

Foram propostas regras de conversão para NoSQL orientado a documentos, em que um DAG representa a estrutura de um documento com suas respectivas entidades aninhadas e regras de conversão para NoSQL orientado a família de colunas, em que um DAG representa uma tabela composta por famílias de colunas. Com base nessas regras, o Metamorfose lê cada DAG e gera um conjunto de operações para ler os dados do RDB, transformar e persistir em formato NoSQL.

O *framework* Metamorfose foi validado por meio de dois experimentos. No primeiro experimento o Metamorfose foi utilizado em um cenário de transformação de dados relacionais abertos, fornecidos pelo Instituto Nacional de Educação e Pesquisa Anísio Teixeira (INEP). Foram usados dados sobre a educação brasileira, incluindo número de matrículas, professores, escolas e cursos disponibilizados em arquivos CSV que ultrapassam 69 milhões de registros por ano. Geralmente, esses dados sofrem variações no formato a cada versão disponibilizada pelo governo, como adição de novos campos, alteração e remoção de campos existentes. O *framework* foi utilizado para mapear e transformar os dados em formato CSV para um esquema relacional pré-definido. Os resultados do experimento foram publicados na trilha demo do SBBD'18 (Kuszera et al., 2018) e mostraram a viabilidade do *framework* em cenários com dados de tamanho moderado.

No segundo experimento o Metamorfose foi usado para migrar dados de um RDB para bases NoSQL orientadas a documentos e a família de colunas. A arquitetura do *framework*, a abordagem para gerar comandos de transformação de dados a partir de um DAG, incluindo a geração de mapeamentos e FDUs foi apresentada. Os resultados obtidos mostraram a eficácia do Metamorfose no processo de conversão e migração de dados e a flexibilidade em termos de

transformação de dados, na qual é possível definir diferentes DAGs para representar a estrutura dos dados que serão migrados para NoSQL. *Framework* e resultados do experimento foram publicados na conferência SAC'19 (Kuszera et al., 2019).

A segunda contribuição desta tese é um conjunto de métricas baseadas em consultas (QBM, do inglês *Query-Based Metrics*), que tem por objetivo medir a cobertura que um determinado esquema NoSQL orientado a documentos fornece para um conjunto de consultas, que representa o padrão de acesso da aplicação (operações de leitura). As métricas são destinadas à fase de modelagem dos dados, antes de executar a migração para NoSQL.

No contexto desta tese, esquema e consulta são representados por meio de DAGs, permitindo comparar a estrutura de ambos por meio de métricas objetivas. Foram definidas seis métricas, em que três são destinadas a medir a cobertura de caminhos, duas destinadas a medir cobertura de arestas e uma para medir o número de coleções de documentos necessárias para responder uma consulta (semelhante a operação de junção em bases relacionais). Por meio das métricas o usuário especialista pode identificar quais consultas são ou não cobertas pelo esquema, quais consultas necessitam relacionar documentos de duas ou mais coleções, a profundidade em que a consulta foi localizada no esquema, ou mesmo quais consultas necessitam de maior atenção. Para avaliar um conjunto de esquemas foram definidos dois escores, chamados *Query Score* e *Schema Score*, usados para ranquear os esquemas e auxiliar o usuário no processo de seleção do esquema mais adequado, conforme requisitos da aplicação. Para calcular os escores é possível atribuir pesos distintos para as métricas, com o objetivo de priorizar determinado tipo de cobertura. De forma semelhante, é possível atribuir pesos distintos para as consultas, com o objetivo de priorizar certas consultas (consultas executadas com maior frequência, por exemplo). Também foram apresentadas diretrizes para implementar as consultas com base na estrutura do DAG, considerando como os documentos estão relacionados no esquema NoSQL. Essas diretrizes foram identificadas durante o processo implementação das consultas e constituem um conjunto de boas práticas, além de padronizar o processo de implementação.

Para validar o conjunto de métricas foram realizados dois experimentos. O primeiro experimento avaliou um conjunto de esquemas NoSQL criados a partir de regras de conversão de RDB para NoSQL extraídas do estado da arte. Os esquemas NoSQL foram avaliados e ranqueados em relação a um conjunto de consultas, priorizando esquemas com maior cobertura ao padrão de acesso da aplicação. Os resultados das métricas foram comparados com duas métricas relacionadas a implementação das consultas (LoC e número de estágios). Para tal, os esquemas NoSQL foram usados para migrar os dados do RDB para uma base MongoDB, usando o Metamorfose para transformar os dados. Na sequência, todas as consultas foram implementadas manualmente por meio do *framework* MongoDB *Aggregation Pipeline*, conforme diretrizes definidas no Capítulo 7. A comparação entre os resultados das métricas, LoC e número de estágios mostrou que esquemas mais próximos ao padrão de acesso das consultas apresentam menor esforço de implementação das consultas. Esse aspecto é um importante indicador e está relacionado ao esforço de manutenção da aplicação. A definição das métricas e resultado dos experimentos foram publicados na conferência CAiSE'20 (Kuszera et al., 2020b). Na mesma conferência foi publicado um artigo de demonstração apresentando a ferramenta *QBMetrics*, também desenvolvida nesta tese (Kuszera et al., 2020a). A ferramenta fornece suporte ao processo de conversão, incluindo a definição de esquemas NoSQL, conjunto de consultas de entrada e cálculo das métricas e escores.

O segundo experimento avaliou o impacto da localização do filtro da consulta no esquema. Foram definidos quatro esquemas NoSQL representando possíveis formas de aninhar três tabelas de um RDB, e seis consultas em que o filtro estava localizado em diferentes entidades do esquema. Os esquemas foram usados para migrar os dados do RDB para o MongoDB e as

consultas foram implementadas, de maneira semelhante ao experimento anterior. Os resultados mostraram que a localização do filtro tem forte impacto na implementação das consultas, principalmente quando o filtro está localizado em campo *array* de documentos embutidos no esquema. Quando isso ocorre é necessário um número maior de estágios e LoC para implementar a consulta, mesmo se o esquema apresenta alta correspondência estrutural com o padrão de acesso da consulta.

Os resultados obtidos mostraram a eficácia das métricas no processo de avaliação de esquemas NoSQL candidatos, auxiliando o usuário na seleção do esquema antes de executar o processo de migração de dados. Os experimentos também mostraram a relação entre os resultados das métricas e o esforço de implementação das consultas em bancos de dados NoSQL orientados a documentos, considerando a correspondência estrutural entre esquema e consulta, e localização do filtro da consulta no esquema.

Como trabalhos futuros, pretende-se estender o *framework* Metamorfose e o conjunto de métricas. Abaixo são listadas possibilidades de extensão e melhorarias na abordagem de conversão apresentada nesta tese:

- O Metamorfose pode ser estendido com novas FDUs para transformação de dados em diferentes formatos. Pretende-se adicionar novas FDUs para possibilitar ao usuário especialista diferentes estratégias de conversão para representar o relacionamento entre as entidades.

- Pretende-se também, realizar avaliação de desempenho do Metamorfose em relação a abordagens de conversão existentes, como *Mongify*, *Transporter* e *NotaQL*. É interessante verificar o comportamento e desempenho do Metamorfose em cenários de conversão de grandes bases de dados relacionais para NoSQL.

- O conjunto de métricas apresentado não considera o custo de execução das consultas no banco de dados. As métricas são destinadas à fase de modelagem do banco de dados NoSQL. Como trabalho futuro pretende-se estender o conjunto de métricas, com o objetivo de integrar com abordagens que consideram o custo de execução das consultas. Para tal, será necessário estender a abstração usada para representar um esquema NoSQL para suportar informações acerca do tamanho do banco de dados (número de instâncias, atributos, cardinalidade, etc). Pretende-se também adicionar métricas para estimar o impacto da localização do filtro da consulta no esquema.

- Nos experimentos para validar as métricas foram considerados apenas bancos de dados NoSQL orientados a documentos. Como trabalho futuro pretende-se adaptar as métricas e experimentos para avaliar bancos de dados NoSQL orientados a famílias de colunas. Demais categorias de bancos de dados NoSQL também serão investigadas.

- Nos experimentos realizados as consultas SQL foram convertidas para DAGs e implementadas manualmente usando o MongoDB *Aggregation Pipeline*. Como trabalho futuro pretende-se investigar formas de conversão automática das consultas, com o objetivo de gerar o conjunto de operadores do *pipeline* automaticamente, facilitando a migração das consultas para o banco de dados NoSQL.

REFERÊNCIAS

Abdelhedi, F., Ait Brahim, A., Atigui, F. e Zurfluh, G. (2017). MDA-Based Approach for NoSQL Databases Modelling. Em Bellatreche, L. e Chakravarthy, S., editores, *Big Data Analytics and Knowledge Discovery*, páginas 88–102, Cham.

Alotaibi, O. e Pardede, E. (2019). Transformation of Schema from Relational Database (RDB) to NoSQL Databases. *Data*, 4(4):148.

Arnicans, G. e Janis, B. (1998). Application generation for the simple database browser based on the ER diagram. Em *Databases and Information Systems, Proceedings of the Third International Baltic Workshop*, volume 1, páginas 198–209.

Astrahan, M. e Chamberlin, D. (1975). Implementation of a Structured English Query Language. *Communications of the ACM*, 18(10):580–588.

Bugiotti, F., Cabibbo, L., Atzeni, P. e Torlone, R. (2014). Database Design for NoSQL Systems. Em Yu, E., Dobbie, G., Jarke, M. e Purao, S., editores, *Conceptual Modeling*, páginas 223–231, Cham. Springer International Publishing.

Cherfi, S. S.-S., Akoka, J. e Comyn-Wattiau, I. (2003). Conceptual Modeling Quality - From EER to UML Schemas Evaluation. Em Spaccapietra, S., March, S. T. e Kambayashi, Y., editores, *Conceptual Modeling — ER 2002*, páginas 414–428, Berlin, Heidelberg. Springer Berlin Heidelberg.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

Davoudian, A., Chen, L. e Liu, M. (2018). A Survey on NoSQL Stores. *ACM Comput. Surv.*, 51(2).

Dean, J. e Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113.

Di Tria, F., Lefons, E. e Tangorra, F. (2017). Cost-benefit analysis of data warehouse design methodologies. *Information Systems*, 63:47–62.

Doan, A., Halevy, A. e Ives, Z. (2012). *Principles of Data Integration*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

dos Santos Ferreira, G., Calil, A. e dos Santos Mello, R. (2013). On Providing DDL Support for a Relational Layer over a Document NoSQL Database. Em *Proceedings of International Conference on Information Integration and Web-Based Applications & Services*, IIWAS '13, página 125–132, New York, NY, USA. Association for Computing Machinery.

El Alami, A. e Bahaj, M. (2016). Migration of a relational databases to NoSQL: The way forward. Em *2016 5th International Conference on Multimedia Computing and Systems (ICMCS)*, páginas 18–23.

Elmasri, R. e Navathe, S. (2011). *Sistemas de Bancos de Dados*. Addison Wesley.

Freitas, M. C. d., Souza, D. Y. e Salgado, A. C. (2016). Conceptual Mappings to Convert Relational into NoSQL Databases. Em *Proceedings of the 18th International Conference on Enterprise Information Systems*, ICEIS 2016, página 174–181, Setubal, PRT. SCITEPRESS - Science and Technology Publications, Lda.

Gómez, P., Roncancio, C. e Casallas, R. (2018). Towards Quality Analysis for Document Oriented Bases. Em *Conceptual Modeling*, páginas 200–216, Cham. Springer International Publishing.

Gómez, P., Casallas, R. e Roncancio, C. (2016). Data schema does matter, even in NoSQL systems! Em *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*, páginas 1–6.

Haas, L. M., Hernández, M. A., Ho, H., Popa, L. e Roth, M. (2005). Clio Grows up: From Research Prototype to Industrial Tool. Em *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, página 805–810, New York, NY, USA. Association for Computing Machinery.

IEEE Std 610.12 (1990). IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, páginas 1–84.

Jaffe, D. e Muirhead, T. (2005). The open source dvd store test application. `https://linux.dell.com/dvdstore/`. Acessado em 28/06/2018.

Jia, T., Zhao, X., Wang, Z., Gong, D. e Ding, G. (2016). Model Transformation and Data Migration from Relational Database to MongoDB. Em *2016 IEEE International Congress on Big Data (BigData Congress)*, páginas 60–67.

Karnitis, G. e Arnicans, G. (2015). Migration of Relational Database to Document-Oriented Database: Structure Denormalization and Data Transformation. Em *2015 7th International Conference on Computational Intelligence, Communication Systems and Networks*, páginas 113–118.

Klettke, M., Schneider, L. e Heuer, A. (2002). Metrics for XML Document Collections. Em *Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering*, EDBT '02, páginas 15–28, London, UK, UK. Springer-Verlag.

Kuszera, E. M., Peres, L. M. e Didonet Del Fabro, M. (2020a). QBMetrics: A Tool for Evaluating and Comparing Document Schemas. Em Herbaut, N. e La Rosa, M., editores, *Advanced Information Systems Engineering - CAiSE Forum*, páginas 77–85, Cham. Springer International Publishing.

Kuszera, E. M., Peres, L. M. e Didonet Del Fabro, M. (2020b). Query-Based Metrics for Evaluating and Comparing Document Schemas. Em Dustdar, S., Yu, E., Salinesi, C., Rieu, D. e Pant, V., editores, *Advanced Information Systems Engineering*, páginas 530–545, Cham. Springer International Publishing.

Kuszera, E. M., Peres, L. M. e Fabro, M. D. D. (2018). Metamorfose: a Data Transformation Framework Based on Apache Spark. Em *33rd Annual Brazilian Symposium on Databases: Proceeedings Companion, SBBD 2018 Companion, Rio de Janeiro, RJ, Brazil, August 25-26, 2018.*, páginas 11–16.

Kuszera, E. M., Peres, L. M. e Fabro, M. D. D. (2019). Toward RDB to NoSQL: Transforming Data with Metamorfose Framework. Em *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, página 456–463, New York, NY, USA. Association for Computing Machinery.

Lee, C. e Zheng, Y. (2015). SQL-to-NoSQL Schema Denormalization and Migration: A Study on Content Management Systems. Em *2015 IEEE International Conference on Systems, Man, and Cybernetics*, páginas 2022–2026.

Mecca, G., Papotti, P., Raunich, S. e Santoro, D. (2012). What is the IQ of Your Data Transformation System? Em *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, página 872–881, New York, NY, USA. Association for Computing Machinery.

Mior, M. J., Salem, K., Aboulnaga, A. e Liu, R. (2016). NoSE: Schema design for NoSQL applications. Em *2016 IEEE 32nd ICDE*, páginas 181–192.

Misra, S., FALOLA, O., Damasevicius, R. e Adewumi, A. (2017). Evaluation and Comparison of Metrics for XML Schema Languages. *Frontiers in Artificial Intelligence and Applications*, 295:51–59.

Moody, D. L. (1998). Metrics for Evaluating the Quality of Entity Relationship Models. Em Ling, T.-W., Ram, S. e Li Lee, M., editores, *Conceptual Modeling – ER '98*, páginas 211–225, Berlin, Heidelberg. Springer Berlin Heidelberg.

Pritchett, D. (2008). BASE: An Acid Alternative. *ACM Queue*, 6(3):48–55.

Pusnik, M., Hericko, M., Budimac, Z. e Sumak, B. (2014). XML schema metrics for quality evaluation. *Comput. Sci. Inf. Syst.*, 11:1271–1289.

Sadalage, P. J. e Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition.

Santos, M. Y. e Costa, C. (2016). Data Models in NoSQL Databases for Big Data Contexts. Em Tan, Y. e Shi, Y., editores, *Data Mining and Big Data*, páginas 475–485, Cham. Springer International Publishing.

Scavuzzo, M., Nitto, E. D. e Ceri, S. (2014). Interoperable Data Migration between NoSQL Columnar Databases. Em *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations*, páginas 154–162.

Schildgen, J., Lottermann, T. e De, S. (2016). Cross-system NoSQL Data Transformations with NotaQL. Em *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, BeyondMR '16, páginas 5:1–5:10, New York, NY, USA. ACM.

Serrano, D., Han, D. e Stroulia, E. (2015). From Relations to Multi-dimensional Maps: Towards an SQL-to-HBase Transformation Methodology. Em *2015 IEEE 8th International Conference on Cloud Computing*, páginas 81–89.

Sommerville, I. (2011). *Engenharia de Software*. PEARSON BRASIL.

Stanescu, L., Brezovan, M. e Burdescu, D. D. (2016). Automatic mapping of MySQL databases to NoSQL MongoDB. Em *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, páginas 837–840.

Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N. e Helland, P. (2007). The End of an Architectural Era (It's Time for a Complete Rewrite). Em *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, páginas 1150–1160.

Timóteo, A. L., Álvaro, A., De Almeida, E. S. e de Lemos Meira, S. R. (2008). Software metrics: a survey. *Sl: sn*.

Vajk, T., Fehér, P., Fekete, K. e Charaf, H. (2013). Denormalizing data into schema-free databases. Em *2013 IEEE 4th International Conference on Cognitive Infocommunications (CogInfoCom)*, páginas 747–752.

Xiang Li, Zhiyi Ma e Hongjie Chen (2014). QODM: A query-oriented data modeling approach for NoSQL databases. Em *2014 IEEE WARTIA*, páginas 338–345.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S. e Stoica, I. (2012). Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. Em *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, páginas 2–2, Berkeley, CA, USA. USENIX Association.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S. e Stoica, I. (2016). Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 59(11):56–65.

Zhao, G., Li, L., Li, Z. e Lin, Q. (2014). Multiple Nested Schema of HBase for Migration from SQL. Em *2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, páginas 338–343.

Zhao, G., Lin, Q., Li, L. e Li, Z. (2014). Schema Conversion Model of SQL Database to NoSQL. Em *Proceedings of the 2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 3PGCIC '14, página 355–362, USA. IEEE Computer Society.