Geomar André Schreiner

SQLTOKEYNOSQL: UMA CAMADA PARA MAPEAMENTO DE ESQUEMAS RELACIONAIS E DE OPERAÇÕES SQL PARA BANCOS DE DADOS NOSQL BASEADOS EM CHAVES DE ACESSO

Dissertação submetido ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do Grau de Mestre.
Ronaldo dos Santos Mello
Universidade Federal de Santa Catarina: Prof. Dr.
Denio Duarte
Universidade Federal da Fronteira Sul: Prof. Dr.

Florianópolis

2016

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Schreiner, Geomar André
SQLToKeyNoSQL: Uma camada para Mapeamento de Esquemas Relacionais e de Operações SQL para Bancos de Dados NoSQL baseados em Chaves de Acesso / Geomar André Schreiner ; orientador, Ronaldo dos Santos Mello ; coorientador, Denio Duarte. - Florianópolis, SC, 2016.
79 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Ciência da Computação.

Inclui referências

1. Ciência da Computação. 2. Interoperabilidade. 3. Modelo relacional. 4. NoSQL. 5. SQL nas nuvens. I. Mello, Ronaldo dos Santos. II. Duarte, Denio. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

Geomar André Schreiner

SQLTOKEYNOSQL: UMA CAMADA PARA MAPEAMENTO DE ESQUEMAS RELACIONAIS E DE OPERAÇÕES SQL PARA BANCOS DE DADOS NOSQL BASEADOS EM CHAVES DE ACESSO

Este Dissertação foi julgado aprovado para a obtenção do Título de "Mestre", e aprovado em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 04 de Fevereiro 2016.

Prof. Dr.
Carina Dorneles
Universidade Federal de Santa Catarina

Banca Examinadora:

Prof. Dr.
Ronaldo dos Santos Mello
Universidade Federal de Santa Catarina

Prof. Dr.
Denio Duarte
Universidade Federal da Fronteira Sul

Ronaldo dos Santos Mello
Universidade Federal de Santa Catarina

Carmem Hara
Universidade Federal do Paraná

Carina Dorneles
Universidade Federal de Santa Catarina

Renato Fileto
Universidade Federal de Santa Catarina

AGRADECIMENTOS

Agradeço primeiramente a Deus por me proporcionar saúde, força e coragem nessa caminhada.

Aos orientadores, Prof. Dr. Ronaldo dos Santos Mello e Prof. Dr. Denio Duarte, pelos ensinamentos repassados, pela oportunidade de trabalho, pelo ambiente estimulante, e principalmente pela paciência.

À minha esposa Mariane, que sempre me deu apoio, compartilhou comigo os momentos bons e ruins no decorrer do desenvolvimento deste trabalho. Agradeço seu amor, sua paciência e compreensão.

Aos meus pais, por ensinar-me o valor da honestidade e do trabalho, por viabilizar uma formação familiar sólida e correta, pela compreensão à minha ausência, pela motivação, pelas orações, pelo apoio.

Aos meu colegas do LISA (Laboratório para Integração de Sistema e Aplicações Avançadas) que criaram um ambiente de amizade, integração, e conhecimento. Entre eles em especial aos meus amigos: Lucas de Alencar, Juarez Sacenti, Cleto May, Filipe Silva, Felipe Pinto, Felipe Born e Douglas Klein pelas criticas construtivas, auxílios e companheirismo.

RESUMO

Diversas aplicações atualmente produzem e manipulam um grande volume de dados, denominados *Big Data*. Bancos de dados tradicionais, em particular, os Bancos de Dados Relacionais (BDRs), não são adequados ao gerenciamento de Big Data. Devido a isso, novos modelos de dados têm sido propostos para manipular grandes massas de dados, enfatizando a escalabilidade e a disponibilidade. A maioria destes modelos de dados pertence a uma nova categoria de gerenciadores de dados denominados BDs NoSQL. Entretanto, BDs NoSQL não são compatíveis, em geral, com o padrão SQL e desenvolvedores que utilizam BDRs necessitam aprender novos modelos de dados e interfaces de acesso para produzirem aplicações baseadas em Big Data. Para lidar com esta problemática, abordagens têm sido propostas para o suporte da interoperabilidade entre BDRs e BDs NoSQL. Poucas destas abordagens tem a habilidade de suportar mais que um BD alvo, sendo a maioria restrita a um BD NoSQL. Neste contexto, este trabalho propõe uma abordagem para acesso via SQL para dados armazenados em um SGBD NoSQL baseado em Chave de acesso (chave-valor, orientado a documentos ou orientado a colunas). Para isso, é proposto um modelo canônico hierárquico intermediário para o qual é traduzido o modelo Relacional. Este modelo hierárquico pode ser traduzido para modelos de dados NoSQL orientado a colunas, orientado a documentos ou chave-valor. A tradução das instruções SQL é feita para um conjunto intermediário de métodos baseado na API REST, que são traduzidos para a linguagem de acesso dos BDs NoSQL. Além disso a abordagem possibilita o processamento de junções que não são suportadas pelos BDs NoSQL. Experimentos demostram que a solução proposta é promissora, possuindo um *overhead* não proibitivo e sendo competitiva com ferramentas existentes.

Palavras-chave: Interoperabilidade 1. Modelo relacional 2. NoSQL 3. SQL nas nuvens 4.

ABSTRACT

A lot of applications produce and manipulate today a large volume of data, the so-called Big Data. Traditional databases, like relational databases (RDB), are not suitable to Big Data management. In order to deal with this problem, a new category of DB has been proposed, been most of them called NoSQL DB. NoSQL DB have different data models, as well as different access methods which are not usually compatible with the RDB SQL standard. In this context, approaches have been proposed for providing mapping of RDB schemata and operations to equivalent ones in NoSQL DB to deal with large relational data sets in the cloud, focusing on scalability and availability. However, these approaches map relational DB only to a single NoSQL data model and, sometimes, to a specific NoSQL DB product. This work presents SQLto-KeyNoSQL, a layer able to translate, in a transparent way, RDB schemata as well as SQL instructions to equivalent schemata and access methods for key-oriented NoSQL DB, i.e., databases based on document-oriented, key-value and column-oriented data models. We propose a hierarchical data model that abstracts the key-oriented NoSQL data models, and use it as an intermediate data model for mapping the relational data model to these NoSQL data models. Besides, we propose the translation of a subset of SQL instructions to an intermediate set of access methods based on the REST API, which are further translated, in a simple way, to the access methods of the key-oriented NoSQL DB. Our solution also supports join queries, which is not a NoSQL DB capability. An experimental evaluation demonstrates that our approach is promising, since the introduced overhead with our layer is not prohibitive.

Keywords: Data interoperability 1. Relational model 2. NoSQL 3. SQL in the cloud 4.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Atualmente, vários tipos de aplicações geram um grande volume de dados que são caracterizados por serem heterogêneos e não possuírem esquemas rígidos. Esses dados são comumente denominados de *Big Data*. Dentre as principais características de *Big Data* estão a flexibilidade de representação, a alta velocidade de geração e o grande volume (os 3 Vs)(BERMAN, 2013). Durante décadas, os bancos de dados relacionais (BDRs) atenderam satisfatoriamente os requisitos de gerenciamento de dados de muitas aplicações, proporcionando simplicidade, robustez e desempenho na manipulação dos dados (STONEBRAKER, 2012). Entretanto, BDRs mostram-se ineficientes para o gerenciamento de *Big Data*, devido, por exemplo, ao comprometimento do desempenho com verificações de consistência de dados, processamento de consultas complexas e a representação rígida e fortemente estruturada de seus esquemas (ABADI, 2009).

Para suprir as necessidades impostas pelo processamento e armazenamento de Big Data, novas arquiteturas de gerenciamento de dados estão sendo desenvolvidas (ZHANG; CHENG; BOUTABA, 2010). Estas arquiteturas utilizam os recursos de alta disponibilidade e escalabilidade atrelados ao paradigma de computação na nuvem, paradigma esse baseado na capacidade de processamento e armazenamento de grandes volumes de dados por centros de dados por meio da Internet (ABOUZEID et al., 2009).

Neste contexto, uma nova família de SGBD chamada de NoSQL (*Not only SQL*) foi proposta(STONEBRAKER, 2012). BDs NoSQL oferecem novos modelos de dados capazes de suportar a representação de dados complexos, além de permitirem a definição de esquemas flexíveis (ABADI, 2009). BDs NoSQL não seguem o modelo relacional de representação dos dados. Portanto, a definição e manipulação de dados através da linguagem SQL não é aplicável. Os principais modelos de dados dos BDs NoSQL são aqueles cujo acesso é baseado em chave, conforme detalhado no Capítulo 2: o modelo *orientado a colunas*, o modelo *orientado a documentos* e o modelo *chave-valor*(ATZENI; BUGIOTTI; ROSSI, 2012).

Apesar dos BDs NoSQL serem mais adequados para lidar com *Big Data*, BDRs vem sendo utilizados por décadas para o gerenciamento de dados (SADALAGE; FOWLER, 2012; STONEBRAKER, 2012). Além disso, usuários e desenvolvedores estão mais habituados à manipulação de dados utilizando a linguagem SQL. Por outro lado, BDs NoSQL definem interfaces de acesso mais simples, geralmente utilizando APIs (*Application Programming Interfaces*) específicas. Consultas mais complexas, envolvendo junções, por exemplo, devem ser codificadas manualmente pelos desenvolvedores e

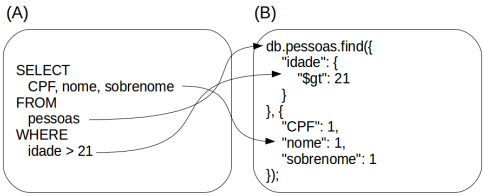

Figura 1 – Exemplo de mapeamento da instrução SELECT da SQL (A) para a linguagem de consulta do MongoDB (B).

tratadas em nível de aplicação (ABADI, 2009).

A Figura 1 apresenta duas consultas: (*a*) uma consulta em SQL e (*b*) uma consulta equivalente em um BD NoSQL orientado a documento (*MongoDB*). As setas demonstram os mapeamentos possíveis de cada uma das partes da consulta SQL. A consulta SQL é convertida em uma busca por um documento *pessoa* em uma coleção de documentos *pessoas*, sendo que o valor do atributo *idade* deve ser maior que ($gt) 21 e o resultado é composto pelos valores dos atributos *CPF, nome* e *sobrenome*.

Esta diferença entre as formas de manipulação dos dados torna complexo o processo de migração para uma solução NoSQL (CHUNG et al., 2013). Nesse contexto, identificam-se três prováveis alternativas para tratar essa problemática. Uma delas é adaptar a aplicação a soluções nativas *relational-in-the-cloud* ou extensões de sistemas gerenciadores de BDR (SGBDRs) comerciais para a nuvem. Esta alternativa, porém, não é interessante devido ao elevado custo de aquisição destas soluções nos dias de hoje. Uma segunda alternativa é a aquisição de um SGBD NoSQL. Entretanto, a curva de aprendizagem e a adaptação das interfaces de acesso para essa nova tecnologia de gerenciamento de dados é também custosa. Por fim, uma terceira alternativa é a criação de uma camada de software que permita o acesso SQL em BDs NoSQL. A dificuldade, neste caso, está no desenvolvimento da camada, porém, entende-se, que o custo associado ao desenvolvimento da camada é menor se comparado com as demais alternativas, pois a aplicação não precisa ser modificada.

Considerando essa terceira alternativa, existem duas abordagens principais na literatura para o desenvolvimento dessa camada: (*i*) adaptar o *kernel* de um SGBDR de forma que possa ser realizada a persistência dos dados em um BD NoSQL (*camada interna*) (EGGER, 2009; ARNAUT; SCHROEDER; HARA, 2011; VILAÇA et al., 2013); (*ii*) desenvolver uma camada independente que seja capaz de receber instruções SQL e executá-las sobre um BD NoSQL (FERREIRA; CALIL; MELLO, 2013; CHUNG et al., 2013; LA-

WRENCE, 2014; RITH; LEHMAYR; MEYER-WEGENER, 2014) (*camada externa*). A segunda abordagem é a adotada nesta dissertação.

Esta dissertação propõe uma camada externa para acesso a BDs NoSQL através de um conjunto restrito de instruções DDL e DML da linguagem SQL. A abordagem é baseada em um modelo canônico hierárquico. Este modelo cria uma abstração para os modelos de dados NoSQL baseados em chave de acesso. A camada, denominada *SQLToKeyNoSQL*, realiza o mapeamento de esquemas relacionais para o modelo canônico proposto, e então, o mapeamento subsequente para os modelos dos BDs NoSQL baseados em chave. Com relação ao mapeamento das instruções SQL, foi proposto um conjunto de três métodos primitivos (*get*, *put* e *delete*) baseados na *API REST* (FIELDING, 2000). As instruções SQL são decompostas em uma ou mais invocações destes métodos primitivos. A execução dos métodos primitivos é feita através de conectores, que implementam o mapeamento dos métodos primitivos para as interfaces de acesso dos BDs NoSQL alvo.

1.1 OBJETIVOS

Esta dissertação possui como objetivo o desenvolvimento de uma abordagem para acesso relacional a dados armazenados em um SGBD NOSQL (orientado a documentos, orientado a colunas ou chave-valor). Para prover este acesso, a abordagem propõe uma camada de abstração que possibilita ao usuário acesso aos dados de forma transparente.

A camada de abstração propõe o uso de um modelo canônico hierárquico que possibilita o mapeamento de um esquema relacional para os modelos de dados NoSQL baseados em chave. Para o mapeamento das instruções SQL são propostos métodos baseados na API REST. Estes métodos são mapeados para as interfaces de acesso dos BDs NoSQL alvo.

A camada possui suporte a um subconjunto de comandos DDL e DML (CREATE TABLE, ALTER TABLE, DROP TABLE, INSERT, UPDATE, DELETE e SELECT). A camada também provê suporte a execução de junções.

Além disso, esta dissertação visa o desenvolvimento de uma arquitetura flexível, onde cada um dos componentes funcione de maneira independente, facilitando, assim, a comunicação de módulos, o que possibilita ganhos de desempenho do sistema.

1.2 CONTRIBUIÇÕES

A principal contribuição desta dissertação é uma abordagem que permite a aplicações manipularem, de forma transparente, dados relacionais mantidos em qualquer BD NoSQL baseado em chave.

Contribuições específicas deste trabalho incluem o modelo canônico proposto, que permite a abstração dos modelos de dados baseados em chaves de acesso. As abordagens *Rith et. al.* (RITH; LEHMAYR; MEYER-WEGENER, 2014) e *Unity* (LAWRENCE, 2014) também são capazes de manipular dados armazenados em diferentes modelos de dados NoSQL, porém, nenhuma destas abordagens utiliza um modelo canônico para realizar o mapeamento. O modelo canônico simplifica o mapeamento, criando uma abstração dos modelos de dados. As abordagens que não utilizam este artifício realizam suas conversões baseados unicamente no mapeamento das instruções SQL sobre a API de acesso do BD NoSQL alvo, tonando-se assim mais dependentes de BDs com linguagens de consulta complexas. *Rith et. al.* realiza o mapeamento para as linguagens de consulta de cada BD NoSQL. Já o *Unity* tem seus mapeamentos inteiramente baseados em conectores (*wrappers*) que são complexos em termos de desenvolvimento e devem ser providos pelo usuário. Apesar da SQLtoKeyNoSQL também depender do desenvolvimento de conectores, como dito anteriormente, seus conectores não são complexos em termos de desenvolvimento, uma vez que apenas implementam os métodos primitivos propostos (*get*, *put* e *delete*) para uso pelos BDs NoSQL.

As regras de mapeamento desenvolvidas nesta dissertação também podem ser citadas como contribuição. As regras propostas incluem o mapeamento do esquema do modelo relacional para o canônico, bem como, o mapeamento do modelo canônico para cada um dos modelos de dados baseados em chave de acesso. Além disso, foi proposto o mapeamento de um subconjunto de instruções SQL para um conjunto de métodos intermediários, posteriormente mapeados para as interfaces de acesso dos BDs NoSQL alvo. Cada um dos trabalhos relacionados possui uma forma própria de realizar seu mapeamento. O diferencial desta dissertação para os demais trabalhos encontrados é a proposta de um modelo de abstração que permite a abordagem suportar mais de um modelo de dados.

Outra contribuição a ser ressaltada é o desenvolvimento de experimentos para avaliação da abordagem proposta. Experimentos com o objetivo de mensurar o *overhead* praticado pela abordagem foram desenvolvidos, bem como experimentos para comparar o desempenho da SQLtoKeyNoSQL com as abordagens SimpleSQL e Unity. Os experimentos demonstram que a abordagem proposta possui um *overhead* não proibitivo e é competitiva se comparada com as demais abordagens.

A abordagem permite que tabelas de um mesmo BDR possam ser armazenadas em BDs NoSQL diferentes. Além disso, ela possui suporte à execução de junções, operação essa não suportada pelos BDs NoSQL. Para prover suporte a essa operação, a arquitetura da SQLtoKeyNoSQL inclui um módulo de processamento das junções que implementa algoritmos clássicos presentes na literatura para execução de junções (MISHRA; EICH, 1992).

1.3 ORGANIZAÇÃO DA DISSERTAÇÃO

Esta dissertação está organizada em mais 5 capítulos. O capítulo 2 apresenta a fundamentação a respeito de Big Data, computação nas nuvens e BDs NoSQL. O capítulo 3 apresenta a proposta desta dissertação, que é a camada SQLtoKeyNoSQL. Este capítulo detalha a sua arquitetura, o modelo canônico e as regras de mapeamento de esquemas e de instruções SQL. O capítulo 4 apresenta a avaliação experimental realizada para verificar a viabilidade da camada. Os trabalhos relacionados são apresentados e comparados no capítulo 5 e as conclusões deste trabalho são apresentadas no capítulo 6.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta conceitos e algumas definições utilizadas para fundamentar a proposta desta dissertação. Inicialmente, são apresentados de maneira mais detalhada os conceitos de Big Data e computação em nuvem. Posteriormente, descrevem-se os BDs NoSQL, suas características e definições dos modelos de dados mais comuns. O modelo relacional, apesar de considerado nesta dissertação, não é detalhado aqui pois é amplamente conhecido na literatura.

2.1 BIG DATA

Big Data é uma denominação para um conjunto de dados que podem ser definidos com os chamados 3Vs (BERMAN, 2013): *(i) Volume* são BDs com grandes quantidades de dados, *(ii) Variedade* diz respeito a dados com heterogeneidade de representação, e *(iii) Velocidade* diz respeito a dados que são gerados com grande frequência e podem estar em constantemente evolução. Para que um conjunto de dados seja considerado Big Data, ele deve atender satisfatoriamente cada um dos 3 Vs.

O gerenciamento de *Big Data* introduz desafios em termos de gerenciamento de dados, uma vez que os convencionais BDRs são ineficazes para o seu tratamento (KUMAR et al., 2014). Para suportar os 3Vs, SGBDs que lidam com Big data são sistemas distribuídos pautados sobre o paradigma de computação na nuvem, facilmente escaláveis horizontalmente, que operam sobre clusters com diversos nodos visando tratar várias requisições de leitura e escrita em paralelo (KLEIN; GORTON, 2015).

2.2 COMPUTAÇÃO NAS NUVENS

Diversos sistemas e arquiteturas estão sendo desenvolvidos para suprir as novas demandas de aplicações com diferentes requisitos de processamento e armazenamento (ABOUZEID et al., 2009). Estes novos sistemas possuem como objetivo fornecer uma nova visão de armazenamento e escalabilidade. Assim, o conceito de computação nas nuvens diz respeito à utilização da capacidade de processamento e armazenamento de grandes centros de dados por meio da Internet (ABOUZEID et al., 2009). O armazenamento de dados é feito através de serviços que podem ser acessados remotamente.

Como principais características da computação nas nuvens pode-se ci-

tar a escalabilidade, o provisionamento dinâmico de recursos sob demanda, a cobrança baseada no uso do recurso ao invés de uma taxa fixa (*pay as you go*) e a distribuição geográfica dos recursos de forma transparente aos usuários (ZHANG; CHENG; BOUTABA, 2010).

Serviços de computação na nuvem são, geralmente, oferecidos em níveis. Alguns níveis a serem citados são (HOFER; KARAGIANNIS, 2011): *(i) IaaS* (Infrastructure as a Service) serviços relacionados ao hardware, processamento de dados, armazenamento e outros; *(ii) PaaS* (Plataform as a Service) serviços relacionados a plataformas, sistemas operacionais, linguagens de programação e ambientes de desenvolvimento; *(iii) SaaS* (Software as a Service) serviços relacionados a execução de programas específicos em diferentes dispositivos; e *(iv) DaaS* (DBs as a Service) serviços relacionados a SGBDs.

O uso de serviços do tipo *DaaS* traz benefícios ao usuário. Um destes benefícios é a redução dos custos na aquisição de um SGBD, delegando ao provedor de serviços a aquisição de licenças e hardware para execução do sistema. Outro benefício é a delegação das tarefas de administração dos dados, como backup dos dados e *tuning* de consultas.

Esta dissertação explora a disponibilidade oferecida pelo *DaaS*. Mais especificamente, esta dissertação propõe o uso de uma nova classe de BDs, os chamados BDs NoSQL, para armazenar dados relacionais. Os BDs NoSQL são apresentados mais detalhadamente na próxima seção.

2.3 BANCOS DE DADOS NOSQL

A justificativa para o surgimento dos BDs NoSQL está relacionada ao advento do *Big Data*. *Big Data* refere-se a uma classe de dados cujas principais características são o seu grande volume, sua variedade de representação e a velocidade com que crescem e devem ser tratados (os chamados 3Vs) (ZIKOPOULOS et al., 2012). Estes dados geralmente são heterogêneos e independentes de um esquema, sendo produzidos por uma gama crescente de aplicações como redes sociais, redes de sensores, bioinformática, dentre outras. O gerenciamento de *Big Data* é um desafio pois exige que requisitos como alta disponibilidade e escalabilidade sejam atendidos por sistemas que manipulam estes dados. Assim, soluções baseadas em computação nas nuvens estão surgindo como um paradigma promissor para lidar com *Big Data* (ABADI, 2009).

BDs NoSQL são, em geral, BDs nas nuvens e visam sanar demandas que os tradicionais SGBDs relacionais não atendem. Eles podem ser definidos como BDs que não seguem o modelo relacional de dados e possuem 6

propriedades (CATTELL, 2011): (*i*) escalonamento horizontal; (*ii*) armazenamento de dados complexos de maneira distribuída; (*iii*) interface de acesso ou protocolo de acesso simples para manipulação dos dados; (*iv*) sem suporte ou relaxamento das propriedades ACID; (*v*) alta disponibilidade; e (*vi*) esquema flexível.

Um modelo de dados é uma combinação de estruturas para organizar um conjunto de dados (ATZENI; BUGIOTTI; ROSSI, 2012). BDs NoSQL suportam, em geral, 4 categorias de modelos de dados (SADALAGE; FOWLER, 2012): (*i*) orientado a documento; (*ii*) orientado a colunas; (*iii*) chave-valor; e (*iv*) orientado a grafos. A abordagem proposta neste trabalho, suporta o mapeamento de um BDR para os 3 primeiros modelos, os chamados BDs NoSQL baseados em chave de acesso.

Os BDs NoSQL baseados em chave de acesso compartilham as noções de manipulação individual de cada item de dados (objetos, registros, etc) e da não rigidez na estrutura de seus itens (ATZENI; BUGIOTTI; ROSSI, 2012). Outro ponto em comum são as interfaces de acesso, que se caracterizam por ser simples, ou seja, permitem basicamente a inserção e exclusão e busca (através de uma chave) de itens individuais de dados. Por outro lado, eles se diferem principalmente na forma pela qual, internamente, é feita a referência aos componentes de um objeto ou registro de dados.

Cada BD NoSQL baseado em chave possui sua própria API de acesso que, em geral, não é compatível com outros BDs NoSQL. Vários desses BDs oferecem suporte à API REST, porém, ela não pode ser considerada um padrão de acesso para esses BDs. A API REST define um gerenciamento de dados unificado através da Web utilizando as interfaces de acesso *get*, *put*, *delete* e *post*.

A próxima seção apresenta os conceitos dos modelos de dados dos BDs NoSQL baseados em chave de acesso.

2.3.1 Modelos de dados baseados em chave

Conforme comentado anteriormente, os modelos de dados NoSQL baseados em chave compreendem o modelo chave-valor, o modelo orientado a documentos e o modelo orientado a colunas. Os conceitos destes modelos de dados são definidos nesta seção, pois são úteis para o detalhamento da camada de mapeamento proposta nesta dissertação.

O modelo de dados chave-valor é o mais mais simples, sendo constituído basicamente por um conjunto de pares chave-valor, onde a chave é utilizada para acessar o valor. O valor pode armazenar um conteúdo simples ou complexo, porém o modelo não faz essa distinção, ou seja, o valor é tratado

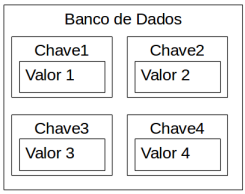

Figura 2 – Exemplo de representação de um BD chave-valor.

como um conteúdo atômico (”caixa-preta”), mesmo que a intenção seja manter um conjunto de valores de diversas propriedades. As consultas em um BD chave-valor se limitam a busca por uma chave, não sendo permitidas consultas que realizam filtros nos valores das chaves. Devido a estas características, assume-se que qualquer valor armazenado em um BD chave-valor possui um domínio atômico. Os conceitos do modelo chave-valor são definidos a seguir.

Definição 1 (Banco de Dados Chave-Valor). *Um BD chave-valor bd é uma tupla* $bd = (n_{bd}, KV_{bd})$*, onde* n_{bd} *é o nome do banco de dados,* KV_{bd} *é um conjunto de pares chave-valor, e bd é acessado pela chave* n_{bd}*.* ◇

Definição 2 (Par Chave-Valor). *Um Par chave-valor* $K_V \in KV_{bd}$ *é definido como* K_V *= (key, value), onde* $K_V.key$ *possui um valor único em* KV_{bd}*, e* $K_V.value$ *armazena um valor atômico.* ◇

A Figura 2 apresenta um exemplo de BD que segue o modelo chave-valor. A única restrição deste modelo é que cada chave associada a um valor tenha um identificador único em todo o BD.

O modelo de dados orientado a colunas organiza dados com base em uma distribuição por colunas (propriedades), esta organização é mais complexa que a anterior e permite consultas com filtro em valores de colunas. Conforme descrito em Sadalage e Fowler (2012), este modelo de dados é composto por uma *keyspace*, famílias de colunas, conjuntos de colunas acessadas com base em uma chave única, colunas e valores. Os conceitos deste modelo de dados são definidos a seguir.

Definição 3 (Keyspace). *Uma keyspace K é uma tupla* $K = (n_K, F)$*, onde* n_K *é o nome de K, F é um conjunto de famílias de colunas, e K acessada pela chave* n_K*.* ◇

Definição 4 (Família de Colunas). *Uma família de colunas* $f \in F$ *é uma tupla* $f = (n_f, S_c)$*, onde* n_f *é o nome de* f*,* S_c *é um conjunto de conjuntos de colunas, e* f *é acessada pela chave* n_f*.* ◇

Definição 5 (Conjunto de Colunas). *Um conjunto de colunas $c_s \in S_c$ é uma tupla $c_s = (n_{c_s}, Cols)$, onde n_{c_s} é o nome de c_s, Cols é um conjunto de colunas, e c_s é acessado pela chave n_{c_s}.* ◇

Definição 6 (Coluna). *Uma coluna $c \in Cols$ é uma tupla $c = (n_c, v)$, onde n_c é o nome de c, v é um valor atômico, e c é acessado pela chave n_c.* ◇

Um exemplo de BD orientado a colunas é apresentado na Figura 3. Neste exemplo é possível perceber uma organização hierárquica dos seus conceitos. Os conjuntos de colunas são identificados por *id*1 e *id*2 e são representados pelos retângulos de borda pontilhada. De acordo com o exemplo, cada identificador de conjunto de colunas pode ter uma quantidade de colunas distinta, por se tratar de itens de dados que podem ter propriedades distintas.

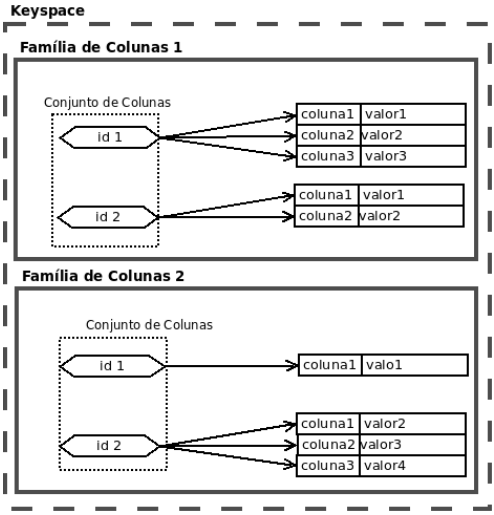

Figura 3 – Exemplo de representação de um BD orientado a colunas.

Por fim, o modelo de dados orientado a documento utiliza o conceito de coleção de documentos, onde cada documento é acessado também a partir de uma chave única e atômica. Da mesma forma que um objeto em um BD orientado a objetos, um documento é composto por uma série de atributos, cujo valor pode ser simples ou complexo. Considera-se um atributo simples aquele que possui um valor atômico e um atributo complexo aquele que possui um conteúdo multivalorado ou um conteúdo organizado em uma estrutura como uma lista, um registro ou um conjunto.

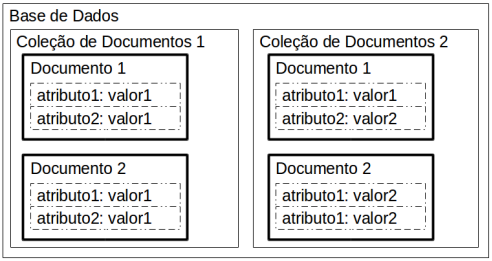

Figura 4 – Exemplo de representação de um BD orientado a documentos.

O modelo de dados de um BD orientado a documento é basicamente composto por um BD, um conjunto de coleções de documentos, documentos, atributos e valores. Cada um destes conceitos é definido a seguir.

Definição 7 (Banco de Dados Orientado a Documentos). *Um BD orientado a documentos D é uma tupla* $D = (n_D, \mathscr{C})$*, onde* n_D *é o nome de D,* $\mathscr{C}$ *é um conjunto de coleções de documentos, e D é acessado pela chave* n_D*.* ◇

Definição 8 (Coleção de Documentos). *Uma coleção de documento* $C \in \mathscr{C}$ *é uma tupla* $C = (n_C, Docs)$*, onde* n_C *é o nome de C, Docs é um conjunto de documentos, e C é acessada pela chave* n_C*.* ◇

Definição 9 (Documento). *Um documento* $d \in Docs$ *é uma tupla* $d = (n_d, A)$*, onde* n_d *é o nome de d, A é um conjunto de atributos, e d é acessado pela chave* n_d*.* ◇

Definição 10 (Atributo). *Um atributo* $a \in A$ *é uma tupla* $a = (n_a, v)$*, onde* n_a *é o nome de a, v armazena um conteúdo atômico ou multivalorado ou um grupo de valores organizado em uma lista, tupla ou conjunto, e a é acessado pela chave* n_a*.* ◇

A Figura 4 apresenta um BD que segue o modelo orientado a documento, onde é possível perceber a organização hierárquica das suas definições, desde o nível mais externo (BD) até o nível mais interno (atributos).

As definições apresentadas nesta seção servem de base para a definição do modelo canônico proposto neste trabalho, conforme descrito na seção 3.1.

2.4 CONSIDERAÇÕES FINAIS

Com o surgimento do *Big Data* torna-se um desafio o gerenciamento de dados. Os SGBDs que lidam com este tipo de dado devem possuir como características o suporte a dados heterogêneos, alta disponibilidade e escalabilidade horizontal. BDs NoSQL são uma nova geração de SGBDs baseados no paradigma da computação na nuvem que surgiram para sanar demandas que os SGBDRs não atendem. Estes BDs propõem novos modelos de dados capazes de suportar a representação de dados complexos na nuvem. Usualmente, seus modelos de dados podem ser classificados em: *chave-valor, orientado a coluna, orientado a documento* e *orientado a grafo*.

Os modelos de dados *chave-valor, orientado a coluna* e *orientado a documento* possuem em comum o fato de possuírem estruturas internadas flexíveis e organizadas hierarquicamente, além de possuírem o acesso de suas estruturas baseados em chaves. Como os BDs NoSQL não seguem o modelo relacional de armazenamento o padrão de consulta SQL não é aplicável a eles, e cada BD possui sua própria API de acesso. No caso dos BDs baseados em chave de acesso, suas estruturas de acesso se assemelham na granularidade das operações, sendo possível armazenar, deletar e consultar um registro de cada vez.

Este capítulo apresentou conceitos que servem de base para a abordagem SQLtoKeyNoSQL. Com base nas definições de cada modelo de dados baseado em chave de acesso foi proposto o modelo canônico, que permite a abstração dos três principais modelos de dados dos BDs NoSQL. A próxima seção apresenta com detalhes a abordagem proposta.

3 PROPOSTA

BDs NoSQL possuem como principais características a alta disponibilidade e escalabilidade horizontal. BDs NoSQL baseados em chave possuem em comum a forma de acesso aos dados, ou seja, qualquer item de dado armazenado é recuperado a partir de sua chave, e a flexibilidade que sua estrutura interna apresenta. Apesar dessas vantagens dos BDs NoSQL no que tange ao gerenciamento de *Big Data*, eles não seguem o modelo relacional de dados e não suportam, em geral, a linguagem de acesso SQL. Porém, o modelo relacional e a SQL são, ainda hoje, amplamente adotados por aplicações que necessitam manter e manipular grandes quantidades de dados. Assim sendo, este trabalho propõe uma abordagem que permite a execução de um subconjunto de instruções SQL sobre BDs NoSQL com acesso baseado em chave, de modo a facilitar a manipulação de dados relacionais quando a intenção é que esses dados sejam portados para a nuvem e mantidos em um BD NoSQL.

A abordagem proposta, denominada SQLtoKeyNoSQL, define uma camada externa de mapeamento que provê o acesso relacional transparente a BDs NoSQL baseados em chave. Essa transparência é garantida principalmente através do mapeamento de um esquema relacional para um esquema correspondente em um modelo canônico hierárquico intermediário que abstrai os 3 modelos de dados NoSQL baseados em chave descritos no capítulo anterior. Estes modelos de dados podem ser sumarizados em 2 conceitos: *chaves* e *valores*. O modelo canônico proposto (seção 3.1) representa chaves e valores de forma hierárquica, provendo um mapeamento simplificado para as representações de dados dos BDs NoSQL. A camada suporta ainda o mapeamento de instruções SQL para um conjunto de métodos baseado na API REST. Esta API provê acesso genérico a grande parte dos BDs NoSQL baseados em chave (FIELDING, 2000).

As próximas seções detalham a proposta. Inicialmente é apresentado o modelo canônico seguido das regras de mapeamento e da arquitetura da camada.

3.1 MODELO CANÔNICO

O modelo canônico é constituído de conjuntos de chaves e valores, organizados hierarquicamente em uma estrutura de árvore, que são capazes de representar o esquema de um BDR. O modelo canônico proposto tomou como base o modelo de representação XML usado para representar um BDR (BOS; W3C, 1997). A estrutura hierárquica do modelo canônico possui 3

níveis a partir da raiz. Os nodos pertencentes a esses níveis definem chaves e os nodos folha abaixo do terceiro nível representam valores atômicos de cada atributo. Como a proposta do modelo canônico é ser um modelo simples e adequado à representação de organizações chave-valor dos modelos de dados NoSQL considerados, evitou-se utilizar outros modelos hierárquicos de dados de propósito geral, como a representação em XML e JSON. Esses modelos possuem diversos conceitos e interfaces de acesso que gerariam processamentos adicionais na manipulação dos dados. A Definição 11 apresenta a estrutura de um esquema de dados representado no modelo canônico.

Definição 11 (Esquema Canônico). *Um esquema Can representado no modelo canônico é um conjunto de chaves organizado hierarquicamente na forma* $Can = (n_{root}(K_{N1_1}, \ldots, K_{N1_n}))$*, onde* n_{root} *é o nodo que denota o nome do BDR,* $(K_{N1_1}, \ldots, K_{N1_n})$ *um conjunto de chaves de Nível 1, e n o número de relações do BDR mapeado.* ◇

Definição 12 (Chave de Nível 1). *Uma chave de Nível 1* K_{N1} *é uma tupla* $(K_{N1_i}, (K_{N2_1}, \ldots, K_{N2_q}))$*, onde* K_{N1_i} *é o nome de um nodo que indica uma relação mapeada,* $(K_{N2_1}, \ldots, K_{N2_q})$ *um conjunto de chaves de Nível 2, e q o número de tuplas que a relação mapeada possui.* ◇

Definição 13 (Chave de Nível 2). *Uma chave de Nível 2* K_{N2} *é uma tupla* $(K_{N2_i}, (K_{N3_1}, \ldots, K_{N3_r}))$*, onde* K_{N2_i} *é um nodo que indica uma tupla de* K_{N1} *nomeada através da concatenação dos valores de cada um dos atributos que definem a chave primária de* K_{N1} *(se a tabela não possuir uma chave primária, a chave de acesso será obtida pela concatenação dos valores de todos os atributos da tupla*[1]*),* $(K_{N3_1}, \ldots, K_{N3_r})$ *é um conjunto de chaves de Nível 3, e r o número de atributos que* K_{N2_i} *possui.* ◇

Definição 14 (Chave de Nível 3). *Uma chave de Nível 3* K_{N3} *é organizada na forma* (K_{N3_i}, v)*, onde* K_{N3_i} *é um nodo que indica o nome de um atributo de uma chave de nível 2 e cujo nodo filho mantém o seu valor* v*.* ◇

Percebe-se, pelas definições, que no modelo canônico a raiz representa o nome do BDR mapeado, cada uma das chaves de nível 1 refere-se a uma relação do BDR mapeado, já as chaves de nível 2 representam os identificadores para cada tupla da relação mapeada e, finalmente, as chaves de nível 3 representam os atributos e os respectivos valores para cada uma das tuplas mapeadas nas chaves de nível 2.

A Figura 5 mostra um exemplo de esquema no modelo canônico (Figura 5 (B)) que representa o mapeamento de um BDR *Cinema* (Figura 5 (A)),

[1] A chave é o identificador único de uma tupla relacional.

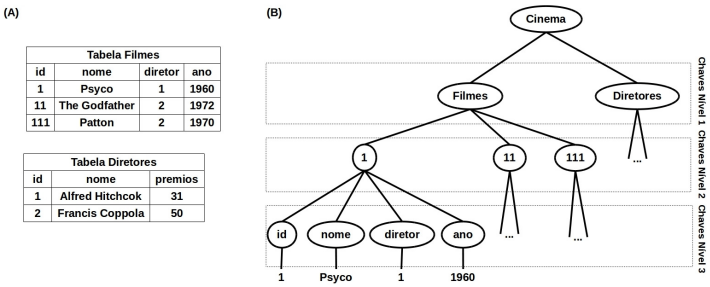

Figura 5 – Exemplo de Representação de um BDR (A) no Modelo Canônico (B).

conforme descrito na Definição 11. O nome *Cinema* é atribuído a n_{root}. As tabelas *Filmes* e *Diretores* são mapeadas para chaves de primeiro nível, as chaves primárias de ambas as tabelas (atributos *id*) são mapeadas para as chaves de segundo nível, as colunas são mapeadas para chaves de terceiro nível e o último nível contém os valores de cada coluna. Cada uma das tabelas do BDR *Cinema* possui como chave primária apenas o atributo *id*. Porém, caso as tabelas possuíssem chaves primárias compostas, cada chave de segundo nível seria o resultado da concatenação dos valores de cada um dos atributos que fizessem parte desta chave composta. Chaves estrangeiras não possuem uma representação específica no modelo canônico, sendo tratadas igualmente como atributos, como é o caso do atributo *diretor* da tabela *Filmes*. A informação que um atributo ou conjunto de atributos atua como uma chave estrangeira é mantida apenas no dicionário de dados da camada (ver seção 3.3.1 para maiores detalhes). Em suma, um esquema relacional no modelo canônico é visto com uma estrutura em árvore, onde o nodo raiz indica o BDR e os nodos folha os conteúdos de dados.

3.2 ESTRATÉGIAS DE MAPEAMENTO

A camada *SQLToKeyNoSQL*, de modo a prover a correspondência relacional-NoSQL, suporta dois tipos de mapeamento entre modelos de dados: (i) mapeamento em nível de esquema; e (ii) mapeamento das instruções SQL. O mapeamento em nível de esquema é regido por um conjunto de regras e ocorre inicialmente entre um esquema relacional e um esquema no modelo canônico proposto, seguido do mapeamento do esquema canônico para esquemas em cada um dos modelos de dados BDs NoSQL baseados em chave.

O mapeamento relacional-canônico é descrito pela Definição 11. A próxima subseção detalha os mapeamentos específicos canônico-NoSQL para cada um dos modelos NoSQL baseados em chave.

O mapeamento de instruções SQL considera um subconjunto das principais instruções DDL e DML desta linguagem. A subseção 3.2.2 detalha quais são as capacidades suportadas para as instruções consideradas pela SQLtoKeyNoSQL e os mapeamentos definidos para a API REST.

3.2.1 Mapeamento do Modelo Canônico para BDs NoSQL

Conforme salientado anteriormente, todos os modelos de dados NoSQL baseados em chave podem ser vistos, de maneira abstrata, como conjuntos de chaves e valores, simplificando, assim, o mapeamento do modelo canônico para cada um destes modelos.

Nesta seção são apresentadas as regras de mapeamento para cada um dos três modelos de dados baseados em chave de acesso. No caso do modelo chave-valor, que apresenta um modelo de dados simples, o mapeamento proposto baseia-se unicamente na ideia do agrupamento dos dados de cada uma das tuplas. Assim, valores de atributos de uma mesma tupla são armazenados como se fossem um valor único. Já os modelos de dados orientado a colunas e orientado a documentos, por serem modelos de dados mais complexos, possuem mapeamentos baseados em agrupamentos de tuplas e, posteriormente, um agrupamento em nível de tabela. O detalhamento das regras de mapeamento é apresentado a seguir.

Um BD chave-valor é constituído unicamente de um conjunto de chaves e valores. Cada chave deve ser única e possuir apenas um valor. Não há restrições quando ao conteúdo de um valor, que pode ser desde um conteúdo atômico até um novo conjunto de chaves e valores. A Regra 1 define o mapeamento para este modelo de dados. Para todas as definições é utilizada a função $nome(nodo)$ para simbolizar a recuperação do valor mantido pelo *nodo* no modelo canônico.

Regra 1 (Mapeamento para BD Chave-Valor). *O mapeamento é definido da seguinte forma:* (i) *um esquema canônico Can gera um BD chave-valor bdkey cujo nome é* $nome(n_{root})$*;* (ii) *cada chave key* $\in$ *bdkey é gerada a partir da concatenação de cada uma das chaves de nível 1* $key1 \in n_{root}.K_{N1_m} (1 \leq m \leq n)$ *com cada uma das suas respectivas chaves de nível 2* $key2 \in key_1.K_{N2_r} (1 \leq r \leq q)$*; e* (iii) *o valor de key é um conjunto de pares chave-valor* KV_{bd}*, onde cada par* $K_V \in KV_{bd}$ *é definido como* $K_V = (nome(key3_l), v_l)$ *onde* $key3_l \in key_2.K_{N3_s} (1 \leq s \leq p)$*, i.e.,* K_V *tem como chave o nome de* $nome(key3_l)$ *e seu valor como valor.*

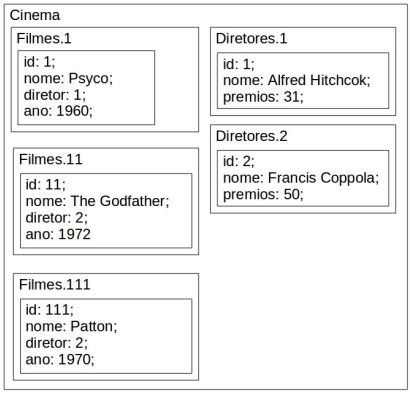

Figura 6 – BD chave-valor resultante da aplicação da Regra 1 para o esquema canônico da Figura 5(B).

A Figura 6 exemplifica a aplicação da Regra 1 para o esquema canônico mostrado na Figura 5(B). O nodo N_{root} (BDR) do esquema canônico é mapeado para o BD *Cinema*. As chaves de nível 1 *key*1 (tabela) são concatenadas com suas respectivas chaves de nível 2 *key*2 (concatenação dos valores da chave primária é realizado), como em $Filmes.1$ e $Diretores.2$, para formarem as chaves. O valor KV_{bd} é representado pelas chaves de nível 3 *key*3 e seus respectivos valores, como $\{id:\ 1;\ nome:\ Psyco;\ diretor:\ 1;\ ano:\ 1960\}$ para a chave $Filmes.1$.

O modelo de dados de um BD orientado a colunas é constituído de *keyspaces*, famílias de colunas, chaves e colunas (SADALAGE; FOWLER, 2012). Uma coluna é constituída de um nome e um valor. Conjuntos de colunas são acessados com base em uma chave. As colunas podem ser agrupadas em famílias, sendo estas compostas por um nome e um conjunto de chaves para acesso às colunas. O número de colunas pode ser variável mesmo para conjuntos que estão em uma mesma família. A Regra 2 define este mapeamento.

Regra 2 (Mapeamento para BD Orientado a Colunas). *O mapeamento é definido da seguinte forma:* (i) *um esquema canônico Can gera um* keyspace $\mathscr{K}$ *cujo nome é* $nome(n_{root})$*;* (ii) *cada chave de Nível 1* $key1 \in n_{root}.K_{N1_m}(1 \leq m \leq n)$ *gera uma família de colunas* $\mathscr{F} \in \mathscr{K}$ *cujo nome é* $nome(key1)$*;* (iii) *cada chave de nível 2* $key2 \in key_1.K_{N2_r}(1 \leq r \leq q)$ *gera uma chave de acesso* $ch \in \mathscr{F}$ *cujo nome é* $nome(key2)$*; e* (iv) *cada chave de nível 3* $key3 \in key_2.K_{N3_s}(1 \leq s \leq p)$ *e seu valor* v *geram, respectivamente, uma coluna* $c \in \mathscr{F}$*, indexada por ch, com nome* $nome(key3)$ *e valor* v.

A Figura 7 exemplifica a utilização da Regra 2 para o esquema canônico da Figura 5(B). O nodo raiz do modelo canônico n_{root} é mapeado para a *keyspace Cinema*. Já as chaves de nível 1 *key*1 são mapeadas para famílias de colunas, como *Filmes* e *Diretores*. Cada uma das chaves de nível 2 *key*3 é mapeada para um identificador de conjunto de colunas, como por exemplo, o identificador 1 na família de colunas *Filmes*. Cada uma das chaves de nível 3 *key*3 com seu valor é mapeada para um conjunto de colunas pertencente a sua respectiva chave de nível 2, como é o caso de $\{$*id:* 1*; nome: Psyco; diretor:* 1*; ano:* 1960$\}$ para o identificador 1 da família de colunas *Filmes*.

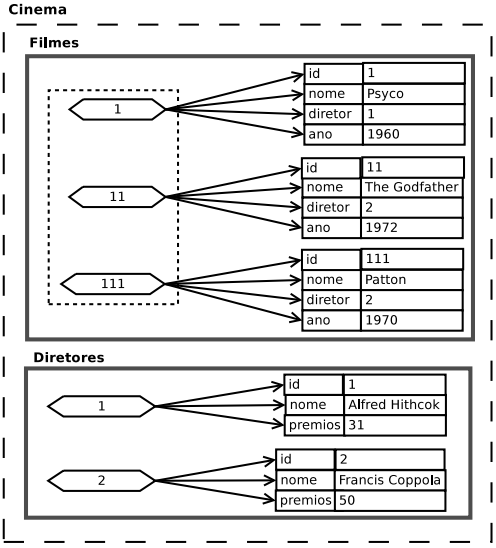

Figura 7 – BD orientado a colunas resultante da aplicação da Regra 3 para o esquema canônico da Figura 5(B).

Por fim, um BD orientado a documento mantém um conjunto de documentos acessados por uma chave com valor atômico. Semelhante a um BD orientado a objetos, o valor de um documento, diferentemente dos BDs chave-valor, não é um conteúdo atômico, mas sim estruturado e composto por atributos simples ou complexos (SADALAGE; FOWLER, 2012). A Regra 3 a seguir define o mapeamento de um esquema canônico para um BD orientado a documentos.

Regra 3 (Mapeamento para BD Orientado a Documento). *O mapeamento é definido da seguinte forma:* (i) *um esquema canônico Can gera um*

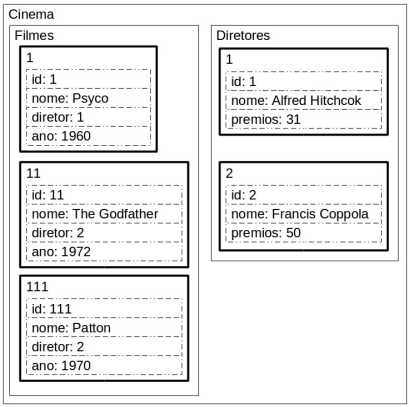

Figura 8 – BD orientado a documento resultante da aplicação da Regra 3 para o esquema canônico da Figura 5(B).

BD orientado a documento bdod cujo nome é $nome(n_{root})$*;* (ii) *cada chave de nível 1* $key1 \in n_{root}.K_{N1_m}(1 \leq m \leq n)$ *gera uma coleção de documentos* $\mathscr{D} \in bdod$ *cujo nome é* $nome(key1)$*;* (iii) *cada chave de nível 2* $key2 \in key_1.K_{N2_r}(1 \leq r \leq q)$ *gera um documento* $d \in \mathscr{D}$ *cuja chave é* $nome(key2)$*; e* (iv) *cada chave de nível 3* $key3 \in key_2.K_{N3_s}(1 \leq s \leq p)$ *gera um atributo* $a_i \in d$ *cujo nome é* $nome(key3)$ *e cujo valor é* v*.*

A Figura 8 apresenta um exemplo da aplicação da Regra 3 para o esquema canônico da Figura 5(B). O nodo n_{root} é mapeado para a BD de documentos *Cinema*. Cada uma das chaves de nível 1 $key1$ (*Filmes* e *Diretores*) é mapeada para uma coleção de documentos $\mathscr{D}$ como *Filmes* e *Diretores*. As chaves de nível 2 $key2 \in$ *Filmes* são mapeadas para documentos d com o mesmo valor de chave, como 1 e 11 em *Filmes*. Já as chaves de nível 3 $key3$ e seus valores são mapeados para atributos do documento e seus respectivos valores, como $\{$ *id:* 1*; nome: Psyco; diretor:* 1*; ano:* 1960$\}$em *Filmes* - 1. Apesar do esquema de representação utilizado nas Figuras 6 e 8 ser bastante similar, cabe ressaltar que os modelos de dados são diferentes, uma vez que o modelo de dados chave-valor não permite consultas com filtros sobre o valor de uma chave, diferente do modelo orientado a documento, onde isso é possível, como por exemplo, buscar os documentos que possuem $ano > 1971$.

As regras de mapeamento apresentadas são utilizadas para o efetivo armazenamento dos dados em BDs NoSQL. Desta forma, o esquema canônico pode ser visto como uma estrutura simples de abstração intermediária. Esta abstração é utilizada para definir um padrão no armazenamento e acesso aos

dados permitindo, assim, que os dados sejam armazenados em BDs NoSQL com modelos de dados distintos, mas possam ser manipulados como se estivessem persistidos em um mesmo modelo de representação, neste caso, o modelo canônico.

Além do mapeamento entre modelos de dados, a *SQLToKeyNoSQL* suporta o mapeamento de instruções SQL visando prover a interoperabilidade de acesso no sentido relacional→NoSQL. O mapeamento das instruções SQL é apresentado com detalhes na próxima seção.

3.2.2 Mapeamento de instruções DDL e DML da SQL

A abordagem SQLtoKeyNoSQL suporta apenas o processamento de um subconjunto de instruções DDL e DML da linguagem SQL para fins de execução correspondente nos BDs NoSQL considerados. Em particular, a abordagem permite o processamento (limitado) das seguintes instruções: CREATE TABLE, ALTER TABLE, DROP TABLE, INSERT, DELETE, UPDATE e SELECT. Para tanto, a camada conta com o apoio de um *dicionário* que mantém informações sobre os esquemas relacionais e os seus mapeamentos para os BDs NoSQL alvo. A estrutura do dicionário é detalhada na Seção 3.3.1. Esta seção detalha os mapeamentos destas instruções.

O mapeamento de instruções DDL utiliza um conjunto de metadados mantido no dicionário da camada que, com o suporte do esquema canônico, define a organização dos BDs NoSQL alvo. As instruções DDL suportadas e os seus tratamentos são os seguintes:

- `CREATE TABLE`: cria as definições da tabela relacional no dicionário. Para esta instrução pode ser informado o nome da tabela, seus atributos e suas restrições de chaves primárias e estrangeiras;

- `ALTER TABLE`: considera-se apenas três ações: modificação do nome de uma coluna, criação de uma nova coluna e remoção de uma coluna. Se uma coluna for excluída, informações sobre ela são retiradas do dicionário e todas as ocorrências da chave de nível 3 correspondentes a ela são também removidas. Caso o nome de uma determinada coluna seja alterado, a entrada da coluna no dicionário é alterada, assim como todas as ocorrências da sua respectiva chave de terceiro nível. Por fim, se uma adição de coluna for realizada, informações sobre ela são adicionadas no dicionário. Nesta primeira versão da SQLtoKeyNoSQL, alterações em chaves primárias não são permitidas, tendo em vista que os dados são armazenados com base nesta chave ou na concatenação delas;

- `DROP TABLE`: exclui a informação da tabela no dicionário e remove a chave de nível 1 (por consequência, ao remover a chave de nível 1, todas as chaves de nível 2 e 3 são excluídas em cascata). A remoção de uma tabela só é permitida se ela não tiver relacionamentos com outras tabelas.

Conforme descrito, essas 3 instruções realizam a manipulação de metadados mantidos no dicionário de dados da camada. No caso do ALTER TABLE, cabe salientar que, ao se modificar um atributo, é desencadeado um processo que, além de alterar essa informação no dicionário, propaga essa alteração para todas as tuplas da tabela mapeadas e mantidas nos BDs NoSQL. Da mesma forma, ao se remover um atributo de uma tabela no dicionário, remove-se, em cascata, este atributo de todas as tuplas correspondentes nos BDs alvo.

Com relação ao tratamento de instruções DML, a camada decompõe cada uma em um ou mais métodos da API REST, conforme salientado anteriormente: (i) *Put* (armazena um valor em uma chave); (ii) *Get* (busca o valor a partir de uma chave); e (iii) $Delete$ (exclui uma chave e seu valor). Estes 3 métodos são ditos métodos primitivos já que qualquer instrução DML é decomposta em um conjunto deles. Assim sendo, inicialmente são definidas uma chave canônica e um registro de dados para, em seguida, definir as primitivas de acesso.

Definição 15 (Chave Canônica). *Chave canônica é uma chave obtida da concatenação de uma chave de nível 1* $key1 \in n_{root}.K_{N1_m}(1 \leq m \leq n)$ *com uma chave de nível 2* $key2 \in key_1.K_{N2_r}(1 \leq r \leq q)$ *associada, em um esquema canônico.* ◇

Um exemplo de chave canônica é a chave criada a partir da concatenação das chaves $Filmes$ e 1, resultando em $Filmes.1$.

Definição 16 (Registro). *Registro é um conjunto de pares chave-valor constituído de uma chave de nível 3* $key3 \in key_2.K_{N3_s}(1 \leq s \leq p)$ *e seus respectivos valores.* ◇

Em suma, a definição de um registro é similar a de uma tupla relacional. Por exemplo, para a chave canônica citada no exemplo anterior, o seu registro é $\{id:\ 1;\ nome:\ Psyco;\ diretor:\ 1;\ ano:\ 1960\}$.

Definição 17 (Put). *Put é um método primitivo na forma* $PUT(k_{put}, v)$*, onde* k_{put} *é uma chave canônica e* v *é um registro a ser armazenado no BD NoSQL alvo.* ◇

O método *Put* armazena um registro com base em uma chave canônica. Um exemplo de invocação deste método com base no esquema canônico da Figura 5(B) é $Put(Filmes.2, \{id : 2; nome : RearWindow; diretor : 1; ano : 1954\})$.

Definição 18 (Get). *Get é um método primitivo na forma* $v = GET(k_{get})$*, onde* k_{get} *é uma chave canônica e v um registro resultante da recuperação de* k_{get} *no BD NoSQL alvo.* ◇

O método *Get* realiza uma busca em um BD NoSQL alvo com base em uma chave canônica, retornando o registro correspondente caso a chave exista. Por exemplo, a execução do método $Get(Filmes.1)$, com base no esquema canônico da Figura 5(B), retorna o registro { *id:* 1*; nome: Psyco; diretor:* 1*; ano:* 1960}.

Definição 19 (Delete). *Delete é um método primitivo na forma* $DELETE(k_{del})$*, onde* k_{del} *é uma chave Canônica a ser excluída de um BD NoSQL alvo juntamente com o seu registro.* ◇

O método *Delete* remove um registro de um BD NoSQL alvo a partir de sua chave canônica. Por exemplo, com base no esquema canônico da Figura 5(B), para remover o registro cujo nome é 'Psyco', deve-se invocar $Delete(Filmes.1)$. Desta forma, além da remoção do registro, a chave de nível 2 1 também é removida.

A partir destes 3 métodos primitivos é proposto o método *GetN* (ver Definição 20), que é composta por um conjunto de métodos *get*. O intuito deste método é explorar a capacidade de alguns BDs NoSQL alvo de recuperar informações em bloco a fim de otimizar o acesso aos dados.

Definição 20 (GetN). $GetN(ck_{all}, cf)$ *é um método que retorna um conjunto de dados* $\mathscr{R}$ *identificados por* ck_{all} *e que respeitam o conjunto de filtros* cf *(possivelmente vazio),* ck_{all} *é um conjunto de chaves canônicas dos dados armazenados e* cf *é constituído de uma série de filtros conectados por operadores lógicos organizados em forma de pilha na ordem em que devem ser executados.* ◇

O método *GetN* busca os valores das n chaves canônicas passadas como parâmetro. Assim, a busca por todos os dados da tabela *Diretores*, por exemplo, pode ser realizada através da chamada $getN(\{Diretores.1, Diretores.2\})$, que retorna os registros { *id:* 1*; nome: AlfredHithcok; premios:* 31} e { *id:* 2*; nome: FrancisCoppola; premios:* 50}, de acordo com a Figura 5(A).

As definições apresentadas são utilizadas para realizar o mapeamento de instruções SQL DML para o padrão de acesso genérico adequado aos BD NoSQL alvo. Cada uma das instruções SQL de entrada é convertida em um conjunto de um ou mais métodos primitivos. As instruções SQL DML suportadas pela *SQLToKeyNoSQL* e os seus tratamentos são os seguintes:

- `INSERT`: decomposto em uma série de métodos *Put* (um para cada registro a ser inserido) que armazenam os registros no BD NoSQL alvo a partir das informações presentes no esquema canônico e no dicionário da camada. Não há suporte para subconsultas e os valores são inseridos com base nos atributos informados, sendo obrigatório informar valores para os atributos que compõem a chave primária;

- `UPDATE`: decomposto em métodos *Get*, *Delete* e *Put*. A atualização de uma ou várias tuplas é possível através de filtros simples[2];

- `DELETE`: decomposto em métodos *Get* e *Delete*. A exclusão de uma ou várias tuplas é possível através de filtros simples;

- `SELECT`: decomposto em uma série de métodos *GetN*. O retorno é um conjunto de tuplas recuperadas a partir de filtros simples. Há o suporte para junções.

A Figura 9(A) apresenta uma instrução de UPDATE que é convertida para uma ocorrência do método *GetN* e uma do método *Put* (Figura 9(B)). A variável *R* recebe todos os registros de $tabela_1$ que respeitam o filtro. O método *atualizaValores* atualiza cada um dos registros em *R* com os atributos passados por parâmetro. Caso algum registro não possua o atributo, este será adicionado ao registro. Após todos os valores serem atualizados, cada registro atualizado gera uma chamada para o método *delete* para que o antigo registro seja removido, e uma chamada para o método *put* que irá armazenar novamente o registro com base em sua chave de acesso.

A Figura 10 apresenta 2 exemplos genéricos de instruções SELECT. Dada a consulta na Figura 10(A), a abordagem a decompõe em um método *GetN* com os respectivos filtros. BDs NoSQL alvo que possuem suporte para a busca de mais de um registro, com base em suas chaves de acesso, podem utilizar o método *GetN* diretamente. Caso contrário, o método é implementado no conector através de sucessivas chamadas do método *Get* para cada chave.

[2]Filtros simples são filtros que executam apenas comparações utilizando os operadores $=, >, <, !, <=, >=$ e que podem estar conectados através de operadores lógicos *AND* e *OR* executados sobre uma mesma tabela.

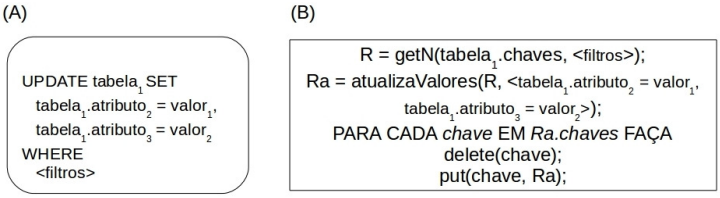

Figura 9 – Exemplo de instrução UPDATE convertida em métodos *GetN*, *Delete* e *Put*.

A consulta da Figura 10(B) é convertida em duas ocorrências do método *GetN*. Como existe uma junção entre duas tabelas, duas chamadas ao método *GetN*, uma para cada uma das tabelas, devem ser realizadas. A projeção dos dados do resultado, embora não apresentada na Figura 10(B), é executada pela camada antes da processamento da junção a fim de diminuir o número de atributos manipulados.

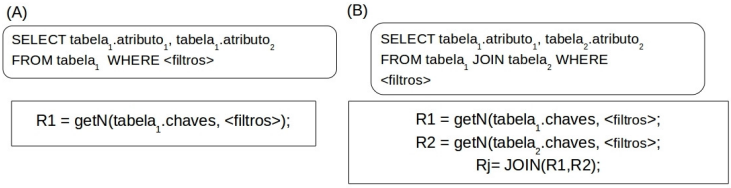

Figura 10 – Exemplo de instruções SELECT convertidas em métodos GetN.

3.3 ARQUITETURA

A arquitetura proposta para a camada *SQLToKeyNoSQL* é constituída de uma *interface de acesso*, um *parser SQL*, um *gerador de plano de consulta*, um *tradutor*, um *módulo de execução*, um *processador de junções*, um *dicionário* e um *modulo de comunicação*. A Figura 11 apresenta a arquitetura da camada. Componentes com linhas pontilhadas são módulos não tratados em profundidade nesta dissertação.

De modo a tornar mais clara a apresentação de cada um dos módulos, a Figura 12 é utilizada como caso de uso. Considera-se, assim, a consulta SQL *Q* (Figura 12(A)) como entrada para a camada.

Inicialmente, *Q* é enviada para o módulo de *Interface de Acesso*. O módulo de *Interface de Acesso* apenas representa a forma de acesso utilizada

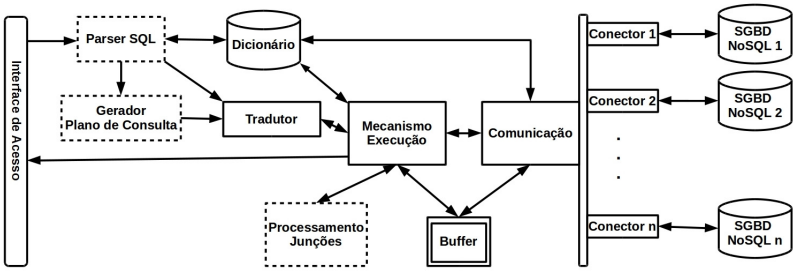

Figura 11 – Arquitetura da camada SQLtoKeyNoSQL.

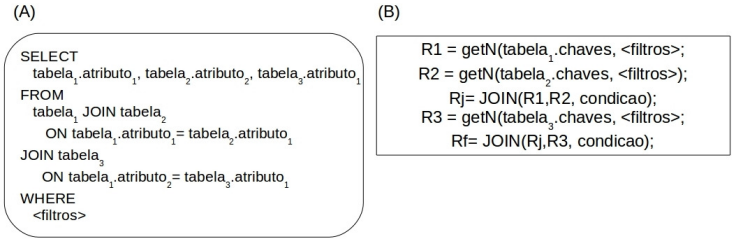

Figura 12 – Caso de uso: SQL com junção de múltiplas tabelas.

pelo usuário. Este acesso, nesta dissertação, é executado através de interface gráfica desenvolvida em *Java*. A *Interface de Acesso* então encaminha o SQL para o *Parser* onde é realizada a verificação sintática e semântica da instrução SQL. Caso a instrução for do tipo DDL ou do tipo `INSERT`, ela é enviada diretamente para o módulo *Tradutor*. Caso for do tipo `SELECT`, `UPDATE` ou `DELETE`, ela é enviada para o módulo *Gerador de Plano de Consulta*. A consulta exemplo Q, neste caso, é enviada para este módulo.

O módulo *Gerador de Plano de Consulta*, pelo fato de não fazer parte do foco deste trabalho, não tem responsabilidade de gerar planos ótimos de consulta. Ele é capaz apenas de otimizar o processamento de consultas que possuem filtros conectados exclusivamente pelo operador lógico *AND* através da sua execução antecipada. No caso do plano de consulta para Q, serão executados inicialmente os filtros presentes na cláusula `WHERE` e, posteriormente, é realizado o processamento das junções. Uma vez gerado o plano de consulta, este é enviado para o módulo *Tradutor*.

O módulo `Tradutor` recebe o plano de consulta ou uma instrução SQL e realiza o seu mapeamento para as interfaces de acesso (*Get*, *Put*, *Delete* e *GetN*) descritos na seção anterior. A tradução do plano

de consulta gerado para Q é mostrada na Figura 12(B).

O módulo *Mecanismo de Execução* recebe os métodos gerados pelo módulo *Tradutor* e os executa por meio do módulo de *Comunicação*. O módulo *Mecanismo de Execução* também é responsável por executar cada filtro e enviar conjuntos de dados ao módulo de *Processamento de Junções*. Considerando Q, o *Mecanismo de Execução* irá executar as operações apresentadas na Figura 12(B). Inicialmente, executa-se o método *GetN* para a $tabela_1$ com a aplicação dos devidos filtros seguido da execução do método *GetN* para $tabela2$. Após a busca dos registros para cada uma destas 2 tabelas, este módulo envia os 2 conjuntos de dados para o módulo de processamento de junções, onde será executada a junção dos dados com base na condição de junção. Neste caso, um *join* nos registros da $tabela1$ que possuem o $atributo_1$ com mesmo valor de $atributo_1$ na $tabela_2$. Uma vez realizado o processamento da junção, o mecanismo de execução busca os registros da $tabela_3$ e então procede a sua junção com o resultado da junção anterior, para, por fim, gerar o conjunto de dados de resposta e encaminhar para o módulo de *Interface de Acesso*. A utilização do *buffer* está condicionada ao tamanho da tabela. Caso a memória principal não comporte os dados a serem manipulados, os dados são acessados através do *buffer*. Os dados mantidos no *buffer* são armazenados em disco através de arquivos e são acessados sequencialmente.

O módulo de *Comunicação* executa as solicitações do módulo *Mecanismo de Execução*, através do envio de métodos para os *Conectores*, e encaminha a resposta obtida dos BDs NoSQL alvo para o mecanismo de execução através do *Buffer*. Os conectores proveem a conexão entre o módulo de comunicação e as APIs de acesso específicas de cada um dos BDs NoSQL alvo. A tarefa de cada conector se resume a realizar pequenas adequações de assinatura da API REST genérica proposta pela camada para as APIs REST específicas dos BDs NoSQL alvo. Caso algum BD NoSQL alvo não possua suporte a API REST, o conector realiza o mapeamento da API REST genérica para a API própria deste BD.

3.3.1 Dicionário

O *dicionário* de dados da SQLtoKeyNoSQLé utilizado para armazenar metadados relativos aos esquemas relacionais de origem (BDs, tabelas, atributos, chaves primárias e chaves estrangeiras) e a localização dos seus dados nos BDs NoSQL alvo. A estrutura do dicionário é definido na sequência.

Definição 21 (Dicionário). *Um dicionário Dic é um tupla* $Dic = (\mathscr{T}, \mathscr{B})$*, onde* $\mathscr{T}$ *é um conjunto de metadados de tabelas (DTable) e* $\mathscr{B}$ *é um conjunto de informações sobre os BDs NoSQL alvo.* ◇

Definição 22 (DTable). *Uma DTable $T \in \mathscr{T}$ é uma tupla $T = (nome, ATT, PK, FK, KEYS, bd)$, onde nome armazena o nome da tabela, $ATT \in$ nome é o conjunto de nomes dos atributos de T, PK é o conjunto de nomes de atributos que define a chave primária de T, $FK = \{(att_1, tnome_1), \ldots, (att_n, tnome_n)\}$ é o conjunto (possivelmente vazio) das chave estrangeiras de T, onde cada par $(atti, tnamei) \in FK$ possui o nome do atributo e o nome da tabela referenciada, $KEYS$ é o conjunto de valores de chaves primárias (chaves de segundo nível no modelo canônico) para cada uma tupla de T, e bd armazena o alias para o BD NoSQL alvo onde os dados de T estão armazenados.* ◇

Definição 23 (BD NoSQL Alvo). *Um BD NoSQL alvo BD é uma tupla $BD = (alias, user, psw, url)$, onde alias é um nome único que identifica o BD, user e psw são, respectivamente, as informações de usuário e senha para acesso a alias, e url é o endereço de acesso para BD.* ◇

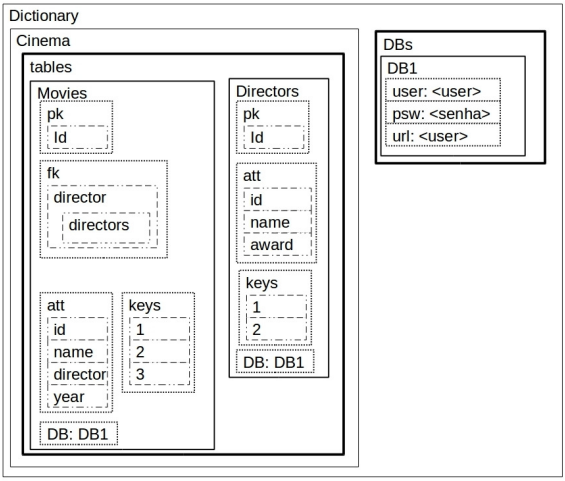

Figura 13 – Examplo de estrutura do dicionário.

A Figura 13 apresenta um exemplo do dicionário de dados para o BDR Cinema (Figure 5(A)). Os metadados para a tabela *Filmes*, por exemplo, são representados pela *DTable Filmes*, que mantém o conjunto de atributos $att : \{id, nome, diretor, ano\}$, o conjunto de chaves primárias $pk : \{id\}$, as chaves estrangeira $fk : \{(diretor, diretores)\}$, as chaves para o acesso de cada um das tuplas $keys : \{1, 2, 3\}$, e o BD NoSQL alvo $BD : BD1$. Percebe-se que, para o gerenciamento das chaves estrangeiras, apenas são armazena-

das no dicionário as informações referentes ao atributo envolvido e a tabela de referência. Assim, o tratamento destas chaves é baseada no conceito de extensão (extended) dos BDs Orientados a Objetos(BERTINO; MARTINO, 1993). A Figura 13 também apresenta os metadados de um BD NoSQL alvo $BD1$.

3.3.2 Processamento de Junções

O processamento das junções é uma tarefa tratada diretamente pela camada SQLtoKeyNoSQL. Inicialmente, o módulo de *Mecanismo de Execução* analisa quais são as junções necessárias, quais os atributos que devem ser combinados e disponibilizados (através do plano de consulta) e quais dados são acessados de forma independente por consultas simples, ou seja, consultas que acessam e aplicam filtros sobre uma única tabela. O retorno de cada uma destas consultas simples é um conjunto de registros, cada um contendo os atributos que o Mecanismo de Execução necessita. Assim sendo, podem haver n conjuntos de dados sobre os quais deverão ser executadas $n-1$ junções. Em pares, este conjuntos de dados são passados pelo Mecanismo de Execução para o módulo de *Processamento de Junções*. Este módulo realiza este processamento aplicando um algoritmo de junção. O retorno deste módulo é um novo conjunto de dados que combina os 2 conjuntos de entrada pelo atributo de junção. Este conjunto é a saída tradicional de uma operação de junção, ou seja, a unificação das colunas de ambas as tabelas e a combinação das linhas que satisfazem o critério de junção. Este conjunto é adicionado aos demais, sendo feita a junção dos dados de uma próxima tabela até que todas as $n-1$ junções tenham sido executadas.

No caso de uso apresentado na seção anterior (Figura 12(A)), a consulta SQL apresenta a junção de 3 conjuntos de dados (n) $tabela_1$, $tabela_2$ e $tabela_3$, onde são executadas 2 junções ($n-1$). O processamento das junções (gerenciado pelo Modulo de Execução) é executado da direita para esquerda, sendo, inicialmente, executada a junção de $tabela_1$ com $tabela_2$, e então o resultado desta primeira junção é utilizado para a execução da junção com $tabela_3$.

O módulo de Processamento de Junções permite que um algoritmo de junção seja facilmente acoplado à abordagem. Ele foi desenvolvido para trabalhar com junções cujos dados possam ser mantidos em memória principal ou em disco utilizando um *Buffer*, conforme apresentado na seção 3.3. O algoritmo de junção executado pelo módulo recebe 3 entradas: 2 conjuntos de dados e a condição de junção. Assim, o desenvolvedor, caso ache necessário, poderá desenvolver seu próprio algoritmo de junção e alterar o algoritmo exis-

tente na abordagem.

Nesta dissertação foram implementados e disponibilizados para a abordagem 3 algoritmos clássicos da literatura: *Nested-Loop Join*, *Sort-Merge Join* e Hash Join (MISHRA; EICH, 1992). Cada algoritmo de junção recebe 2 conjuntos de dados e os atributos nos quais a junção deverá ser executada. O algoritmo *default* para a execução de junções é o *Hash Join* por apresentar o melhor desempenho médio dentre os 3 disponibilizados e os dados são também tratados, por *default*, em memória principal. O algoritmo *Sort-Merge Join* foi implementado a fim de executar junções sobre dados que excedam a capacidade da memória principal. Na atual implementação da SQLtoKeyNoSQLo algoritmo de *join* é alterado pelo usuário, assim, não há um mecanismo que troque o algoritmo automaticamente caso os dados não sejam comportados em memória.

Cabe ressaltar aqui que o foco desta dissertação foi a proposição de uma camada de mapeamento relacional-NoSQL pautada em um modelo canônico e processos de mapeamento. Assim sendo, a proposição de novos algoritmos de junção está fora do escopo deste trabalho. Apenas algoritmos clássicos foram providos a fim de permitir que a SQLtoKeyNoSQLofereça suporte ao processamento de junções devido à ausência desta capacidade pelos SGBDs NoSQL.

3.4 CONSIDERAÇÕES FINAIS

BDs NoSQL possuem como características a alta disponibilidade, escalonamento horizontal e flexibilidade de representação, características essas importantes no tratamento de *Big Data*. Apesar de suas vantagens, BDs NoSQL não possuem uma linguagem padrão de acesso e como não seguem o modelo relacional de dados e não possuem, geralmente, suporte à linguagem SQL. Assim sendo, torna-se pertinente o desenvolvimento de abordagens que realizem o mapeamento relacional-NoSQL.

Este capítulo detalhou a abordagem SQLtoKeyNoSQL, uma camada que permite, de maneira transparente, o armazenamento de dados relacionais e a sua manipulação, através de SQL, em BDs NoSQL baseados em chave de acesso. Para prover tal acesso, é proposto um modelo canônico hierárquico que abstrai os modelos de dados associados a BDs NoSQL baseados em chave de acesso. O modelo canônico é uma estrutura interna de representação que visa mapear, com a ajuda de um dicionário de dados, o esquema de um BDR. As instruções SQL suportadas pela camada são divididas em uma série de operações primitivas baseadas em métodos da API REST. Cada uma destas operações é executada sobre um BD NoSQL alvo através de conectores. Os

conectores apenas implementam as tradicionais operações *Get*, *Put*, *Delete* e, se possível, uma operação *GetN*.

A camada também possui suporte ao processamento de junções, operação esta que não é permitida por BDs NoSQL. Para o processamento de junções foram implementados alguns algoritmos clássicos da literatura. A arquitetura proposta para a camada possui módulos que trabalham de forma articulada, porém cada um deles é independente e pode ser alterado de forma individualizada. Por exemplo, a arquitetura permite que o módulo de Processamento de Junções seja alterado a fim de suportar outro algoritmo de junção, ou ainda, a camada permite a alteração do módulo Gerador de Planos de Consulta a fim de permitir melhores otimizações.

O próximo capítulo descreve um conjunto de experimentos realizados a fim de avaliar a eficácia da camada.

4 EXPERIMENTOS

Este capítulo descreve 2 experimentos realizados sobre a camada SQLtoKeyNoSQLproposta. O primeiro experimento verifica o *overhead* em termos de tempo de execução introduzido pela SQLtoKeyNoSQL para o processamento de instruções SQL DML. Já para as instruções SQL DDL, que não possuem operações semelhantes nos BDs NoSQL, são apresentados os tempos de execução. O segundo experimento tem como objetivo comparar o uso da camada com outras abordagens existentes, mais especificamente, o *SimpleSQL*(CALIL; MELLO, 2012) e o *Unity* (LAWRENCE, 2014). Para cada experimento são apresentados os seus detalhes da execução (domínios dos BDs e configuração do ambiente de execução), seus resultados bem como uma analise dos mesmos. As próximas seções apresentam os experimentos realizados.

4.1 INSTRUÇÕES SQL

Este primeiro experimento avalia o tempo de execução dos instruções DDL e o *overhead* causado pela abordagem no processamento de instruções DML. O *overhead* de instruções DDL não pode ser calculado, pois nem todas executam ações diretas sobre um BD NoSQL alvo. Para a execução do experimento, foi utilizado o BDR *Airbase*[1] que armazena dados referentes a qualidade do ar na Europa.

Apesar do BDR possuir diversas tabelas, somente 2 delas foram consideradas: *station_info*, que armazena informações sobre estações de medição, e *measurement_configuration*, que mantém parâmetros utilizados na medição da qualidade do ar. A Figura 14 apresenta um fragmento do esquema lógico da BDR *Airbase*. A tabela *measurement_configuration* foi escolhida por ser a maior em número de linhas (59040 linhas) e a tabela *station_info* foi escolhida por ser a maior em número de colunas (21 colunas e 8626 linhas).

Uma máquina com processador Intel Core i3-2430M, 6 GB DDR3 1066mHz RAM, rodando o *kernel* Linux 3.19.0-26 (distribuição XUbuntu 15.04) foi utilizada nas execuções deste primeiro experimento. Em termos de BD NoSQL alvo, foram escolhidos um representante de cada BD NoSQL orientado a chave: *Cassandra* (orientado a colunas), *MongoDB* (orientado a documentos) e *Redis* (chave-valor). As instâncias dos BDs NoSQL foram instaladas localmente e configuradas para usarem apenas um nodo de armazenamento. Após a criação dos BDs em cada BD NoSQL alvo, cada instrução

[1]The European Air quality dataBase - http://migre.me/rqSEO

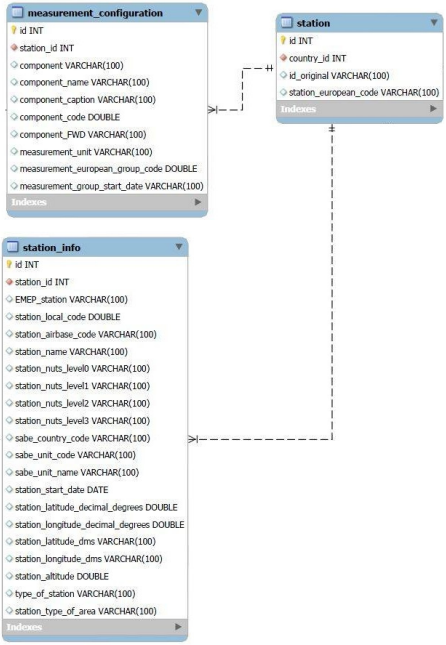

Figura 14 – Esquema parcial do BDR utilizado no primeiro experimento.

SQL proposta foi executada 3 vezes e a média dos tempos de execução foi coletada.

O restante desta seção detalha a execução do experimento bem como a análise dos resultados. Inicialmente, apresenta-se os experimentos relativos às instruções DDL e, na sequência, as instruções DML.

4.1.1 Instruções SQL DDL

BDs NoSQL não possuem, em geral, operações em nível de esquema que sejam comparáveis às instruções SQL DDL dos BDRs. Assim sendo, esta seção apresenta apenas os tempos de execução de cada uma das instruções DDL consideradas pela camada para as tabelas mostradas na Figura 14. A Tabela 1 apresenta os tempos de execução em segundos para cada uma dessas instruções.

Percebe-se, na Tabela 1, que a instrução *CREATE TABLE* apresentou um tempo de execução constante $(0,2)$ para todos os BDs NoSQL e para as

Instruções SQL	*measurement_configuration*			*station_info*		
	Redis	Mongo	Cassandra	Redis	Mongo	Cassandra
CREATE TABLE	0,02 *s*	0,02 *s*	0,02 *s*	0,02 *s*	0,02 *s*	0,02 *s*
ALTER TABLE	2,46 *s*	6,17 *s*	8,62 *s*	0,71 *s*	0,95 *s*	3,07 *s*
DROP TABLE	2,13*s*	4,8 *s*	6,19 *s*	0,43 *s*	0,73 *s*	2,82 *s*

Tabela 1 – Tempo de execução para instruções DDL.

2 tabelas. Este comando não executa operações sobre nenhum dos BDs alvo, restringindo-se a criar a estrutura de cada tabela no dicionário de dados.

A instrução *DROP TABLE* realiza a exclusão de todos os registros de uma tabela, bem como da própria tabela. A exclusão é realizada a fim de evitar a remanescência de registros órfãos[2]. Assim sendo, para cada chave de acesso armazenada no dicionário é deflagrada a exclusão do registro no BD alvo e, após todos os registros serem excluídos, a tabela é excluída do dicionário de dados. Pela tabela percebe-se que cada um dos BDs teve um desempenho diferenciado para a execução desta instrução. O Redis possui o tempo de execução mais baixo e o Cassandra teve o tempo mais alto. O desempenho de cada um dos BDs esta relacionado ao tempo que é necessário para excluir cada registro, já que, para a execução desta instrução, todos os registros são excluídos. O Redis possui um modelo de dados simples e nenhuma estrutura interna relacionada ao valor de cada chave, facilitando assim, a exclusão de chaves e valores. Já o Cassandra é o mais complexo. Ele possui uma série de informações sobre os dados, o que torna a exclusão de seus registros mais onerosa.

A instrução *ALTER TABLE* executada realizou a exclusão de duas colunas de uma tabela. Esta instrução possui o maior tempo de execução dentre todas as instruções DDL, pois para realizar esta operação, inicialmente, é realizada uma operação de busca de todos os registros e, um a um, cada registro é atualizado e armazenado novamente. Após todos serem atualizados, é executada a operação no dicionário de dados. Por exemplo, para realizar a exclusão de um atributo, todos os registros são trazidos para a memória principal, cada registro tem a coluna removida e, posteriormente, é armazenado novamente no BD NoSQL alvo. Da mesma forma que na instrução anterior, o tempo de execução da instrução esta fortemente relacionada à complexidade do modelo de dados. O modelo de dados do Redis, por ser mais simples, propicia a inclusão dos registros diminuindo o tempo de execução da instrução. Já o Cassandra possui estruturas internas que devem ser atualizadas, tornando assim, o processamento desta instrução mais oneroso.

[2]Um registro órfão é um registro que não possui entrada entre as chaves de uma tabela no dicionário de dados ou um registro cuja tabela não existe no dicionário de dados.

4.1.2 Instruções SQL DML

A análise do *overhead* no processamento de instruções DML pela SQLtoKeyNoSQLleva em conta 2 medições de tempo de execução: (*i*) tempo gasto pela camada (t_{layer}) para o processamento das instruções, e (*ii*) tempo gasto pelo BD NoSQL alvo (t_{nosql}) para executar as instruções. Assim, o tempo total de processamento é $t_{total} = t_{layer} + t_{nosql}$.

Para a avaliação da instrução *INSERT* foram definidos 5 *scripts* de inserção de dados em cada uma das tabelas escolhidas. Cada *script* possui um número diferente de inserções a serem executadas a fim de verificar a escalabilidade da abordagem. O número de inserções para a tabela *station_info* foi de 500, 1000, 2000, 4000, e 8000 registros. Já para a tabela *measurement_configuration*, o número de inserções foi de 1000, 5000, 10000, 25000 e 50000 registros. A Figura 15 apresenta o tempo de processamento (t_{total}) da SQLtoKeyNoSQL e o tempo de processamento gasto pelo BD NoSQL alvo (t_{nosql}).

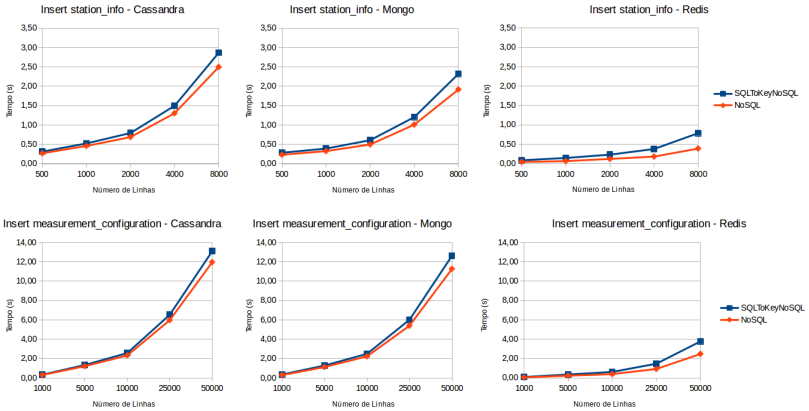

Figura 15 – Tempo de processamento para instruções INSERT.

Os gráficos da Figura 15 revelam que os BDs Cassandra e o Mongo possuem um *overhead* de aproximadamente 12% (t_{layer}/t_{nosql}) no pior caso (8000 linhas) para a tabela *station_info*. Diferentemente, a SQLtoKeyNoSQL tem um *overhead* maior que os demais BDs dobrando o tempo de processamento para a mesma tabela, no caso do BD Redis. Este fato ocorre devido à simplicidade de acesso do BD Redis, *i.e.*, como um representante dos BDs chave-valor, Redis apenas retorna um valor atômico baseado em uma chave. Assim sendo, a SQLtoKeyNoSQL serializa/codifica os dados das

tuplas de forma que esses possam ser tratados como um valor atômico durante seu armazenamento e recuperação. Em uma eventual consulta, os dados são recuperados como um valor atômico, e depois de descodificados são manipulados utilizando a representação do modelo canônico. Apesar do *overhead* maior para o BD NoSQL Redis, percebe-se que este teve o menor tempo de execução.

Já na execução das inserções na tabela *measurement_configuration*, a SQLtoKeyNoSQL utilizando o Cassandra ou o Mongo gastou 13 segundos para armazenar 50k linhas, enquanto havia gasto apenas 3 segundos para a tabela *station_info*. Redis executou todas as inserções em até 3.8 segundos para a tabela *measurement_configuration* e em até 0.8 segundos para a tabela *station_info*. A diferença nos tempos de execução está relacionada à maior complexidade dos modelos de dados do Cassandra e do Mongo, já que o Redis apenas necessita armazenar um valor único dada uma chave.

A fim de verificar a cobertura do mapeamento dos dados, após a execução de cada *script* foi verificado que cada uma das instruções de inserção havia realmente gerado um novo registro (correto) no BD NoSQL alvo.

A Figura 16 apresenta um série de consultas utilizadas na avaliação do *overhead* no processamento de instruções *SELECT* para as tabelas *measurement_configuration* e *station_info*. Para cada uma das tabelas foram definidas 3 consultas com um número crescente de filtros a serem aplicados. Cada consulta foi executada uma primeira vez para fins de "aquecimento"(*warm up*) e, na sequência, foram executadas mais 3 vezes, obtendo-se, assim, a média do tempo de execução destas 3 últimas execuções.

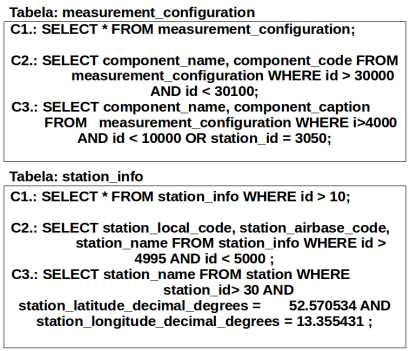

Figura 16 – Consultas consideradas no experimento de instruções SELECT.

A Figura 17 apresenta os tempos de processamento (t_{total} em segun-

dos) para a execução das consultas (Figura 16). Para a validação dos resultados, foram comparados o número de registros retornados pela SQLtoKeyNoSQL com o número de registros retornados pela mesma consulta no BDR PostgreSQL mantendo o mesmo BD *Airbase*. Todos os registros retornados pela SQLtoKeyNoSQLforam também retornados pelo PostgreSQL.

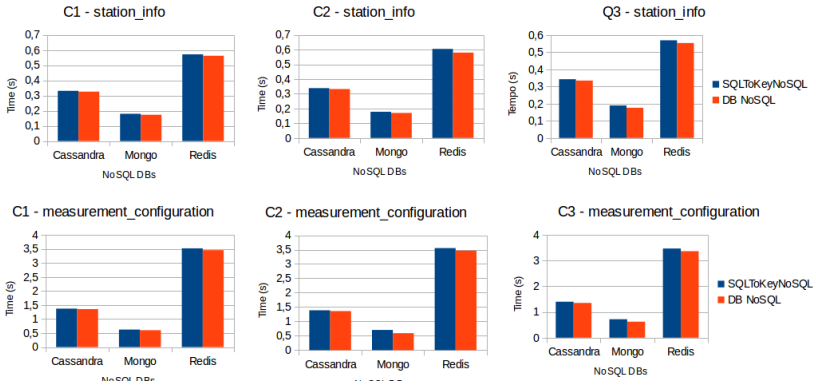

Figura 17 – Gráfico tempo de processamento para as consultas.

De acordo com a Figura 17, o *overhead* ficou abaixo dos 0.3 segundos para todas as consultas, comprovando, assim, que o uso da SQLtoKeyNoSQL não comprometeu, de modo geral, o desempenho na execução das consultas. O BD MongoDB obteve o melhor desempenho dentre os 3 BDs NoSQL alvo. Percebe-se, ainda, que o *overhead* da camada aumenta conforme a complexidade da consulta. Este aumento, embora quase imperceptível, ocorre devido ao processamento dos filtros.

A Figura 18 apresenta um gráfico gerado a partir dos tempos de execução das instruções $UPDATE$ e $DELETE$. A execução de cada uma das instruções teve o objetivo de manipular um único registro em cada uma das tabelas. O filtro utilizado é o descrito na consulta $C3$ da Figura 16 para cada uma das tabelas.

Percebe-se que o *overhead* apresentado pela camada é não proibitivo. O BD Redis apresentou o maior *overhead* e o maior tempo de execução para as 2 instruções. Isso se deve ao fato da simplicidade de seu modelo de dados, exigindo que a camada execute um processamento extra (decodificação dos dados armazenados de maneira atômica) para suportá-lo. O MongoDB foi o BD com melhor desempenho na execução das operações.

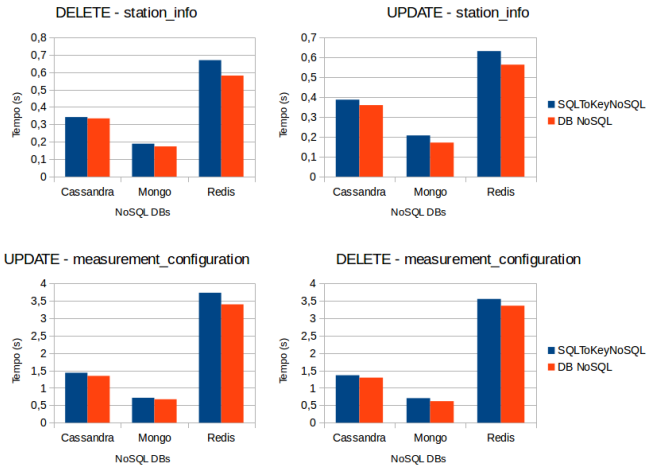

Figura 18 – Gráfico tempo de processamento para as instruções UPDATE e INSERT.

4.2 COMPARAÇÕES COM ABORDAGENS EXISTENTES

O segundo experimento proposto compara o desempenho da abordagem proposta com 2 trabalhos relacionados presentes na literatura: *SimpleSQL* (FERREIRA; CALIL; MELLO, 2013) e o *Unity* (LAWRENCE, 2014). Estes e outros trabalhos relacionados são detalhados no Capítulo 5. O *SimpleSQL* foi escolhido por ser o trabalho mais citado dentre os encontrados. Já o *Unity* foi escolhido por ser um ferramenta comercial[3] e possuir um solução livre para testes.

O SimpleDB e o Unity implementam mapeamentos para BDs NoSQL alvo distintos, apesar de ambas as soluções terem desenvolvido camadas de mapeamento para BDs NoSQL orientados a documentos. Assim, a fim de tornar a competição entre as abordagens mais justa, foram executados 2 experimentos, um para cada uma das abordagens.

O BDR utilizado nestes experimentos diz respeito a dados sobre vestibulares para ingresso na UFSC. A Figura 19 apresenta o esquema lógico do BDR utilizado. O BDR possui 6 tabelas onde são armazenados dados sobre candidatos, suas opções de curso, seus resultados e a qual evento (qual concurso vestibular) eles estão associados.

Uma máquina com processador Intel Core i3-2430M, 6 GB DDR3

[3] `http://www.unityjdbc.com/`

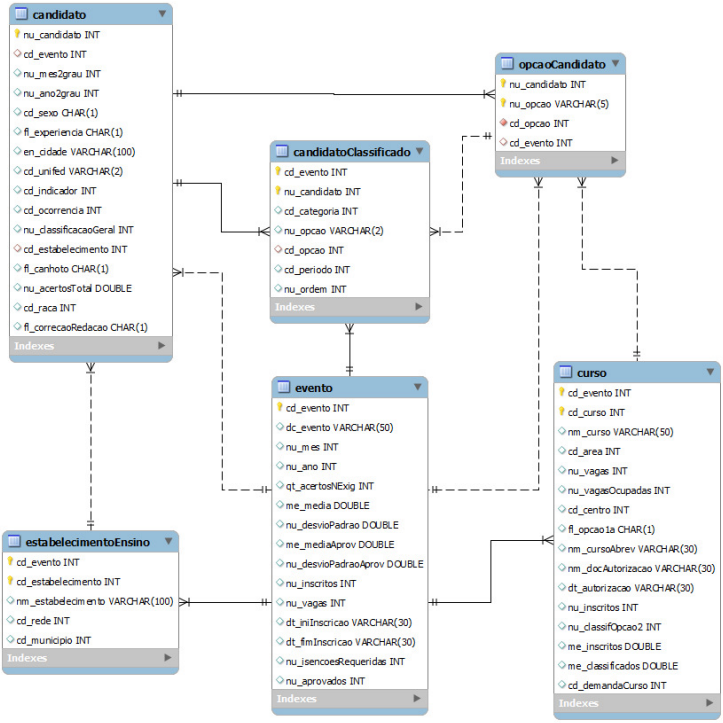

Figura 19 – Esquema do BDR referente a vestibulares da UFSC.

1066mHz RAM, rodando o *kernel* Linux 3.19.0-26 (distribuição XUbuntu 15.04) foi utilizada para a execução dos experimentos. As instâncias dos BDs NoSQL foram instaladas localmente e configuradas para usarem apenas um nodo de armazenamento. Após a criação do BD em cada um dos BDs NoSQL alvos, cada instrução SQL foi executada 3 vezes e a média dos tempos de execução foi considerada.

4.2.1 Experimento SimpleSQL

SimpleSQL é ma camada que possibilita o acesso SQL sobre dados armazenados no SimpleDB. O SimpleDB é um BD NoSQL que implementa um modelo de dados orientado a documento criado e mantido pela Amazon. Por ser um serviço da Amazon, o SimpleDB não pode ser instalado e configurado

localmente. Desta forma, o acesso é feito via Web. Os dados armazenados e acessados durante o experimento foram armazenados de maneira distribuída na região leste do EUA.

Não foi possível executar o SimpleSQL para a execução de novos testes. Assim sendo, foram reproduzidos os mesmos experimentos apresentados em trabalhos que descrevem esta abordagem (Calil e Mello (2012) e Ferreira, Calil e Mello (2013)). O conjunto de dados utilizado nos experimentos desta dissertação é o mesmo utilizado pelos autores dos artigos que propõem o SimpleSQL. O mesmo ambiente de testes utilizado para os experimentos do SimpleSQL foi utilizado para a SQLtoKeyNoSQL. Nos experimentos do SimpleSQL foi utilizado um processador Core i5-2430M, 6 GB DDR3 1066mHz RAM em um sistema operacional Windows 7.

Para a comparação entre as 2 abordagens são considerados apenas alguns comandos SQL: *CREATETABLE*, *DROPTABLE*, *ALTERTABLE*, *INSERT* e *SELECT*. As instruções DDL, bem como as instruções *INSERT*, são executadas sobre a tabela *Candidato* populada com 50000 linhas. A Tabela 2 compara o tempo de execução entre as 2 abordagens para cada uma das instruções.

Instruções SQL	***SimpleSQL***	***SQLtoKeyNoSQL***
CREATE TABLE	47 ms	2 ms
ALTER TABLE	3h 40m 19s	3h 38m 56s
DROP TABLE	3h 27m 37s	2h 40m 21s
INSERT	3h 29m 24s	3h 27m 36s

Tabela 2 – Comparação de tempo de execução entre SimpleSQL e SQLtoKeyNoSQL.

Percebe-se pela Tabela 2 que apenas há diferenças significativas para as instruções *DROP TABLE* e *CREATE TABLE*. No caso da instrução *CREATE TABLE*, como já dito anteriormente, a SQLtoKeyNoSQL não executa operações sobre os BDs alvo, mas apenas sobre o dicionário de dados que é mantido em memória. Por isso, seu custo é bem menor do que o apresentado pelo SimpleSQL que cria estruturas no SimpleDB na execução desta instrução. Já para a instrução *DROP TABLE*, a diferença nos tempos de execução se deve à forma com que os registros são excluídos. Enquanto a abordagem SimpleSQL exclui os registros (mapeados para itens) no SimpleDB através de um filtro pela chave, a SQLtoKeyNoSQL exclui os registros pela chave de acesso, que no caso do SimpleDB, são representadas pela chave de acesso do item. Assim, o SimpleSQL realiza um filtro para exclusão dos itens, enquanto a SQLtoKeyNoSQL exclui o item sem filtros apenas utilizando a chave de acesso do item otimizando a exclusão.

Para as instruções *ALTER TABLE* e *INSERT* a diferença no tempo de

execução não foi tão significativa. Apesar disso, verifica-se que a SQLtoKeyNoSQL possui desempenho superior. A diferença nos tempos de execução de ambas as instruções reside no fato de que, para ambas, o SimpleSQL armazena informações de metadados no SimpleDB (em uma espécie de dicionário de dados) gerando requisições de armazenamento maiores que as apresentadas pela SQLtoKeyNoSQL.

Com relação aos testes com a instrução de seleção, somente 2 consultas foram comparadas, pois outros testes realizados pelo SimpleSQL e que envolvem instruções *SELECT* utilizam a cláusula *LIKE* em filtros e esta cláusula não é suportada pela SQLtoKeyNoSQL. A Figura 20 apresenta os comandos de seleção considerados nos testes.

Tabela: candidato

```
C1.:  SELECT nu_candidato, cd_raca
FROM candidato
WHERE cd_sexo = 'F'


C2.:SELECT
  cc.nu_ordem, c.en_cidade, ee.nm_estabelecimento, e.dc_evento
FROM
  Candidato c INNER JOIN estabelecimentoEnsino ee
    ON c.cd_estabelecimento = ee.cd_estabelecimento
  INNER JOIN candidatoClassificado cc
    ON c.nu_candidato = cc.nu_candidato
  INNER JOIN e ON cc.cd_evento = e.cd_evento
WHERE e.cd_evento = 25
 AND ee.cd_evento = 25
 AND cc.cd_evento = 25
 AND c.cd_evento = 25;
```

Figura 20 – Consultas consideradas no experimento do SimpleSQL.

Consulta	*SimpleSQL*	*SQLtoKeyNoSQL*
C1	3,15 min	0.96 min
C2	24,85 min	13,39 min

Tabela 3 – Comparação do tempo de execução entre SimpleSQL e SQLtoKeyNoSQLpara consultas.

A Tabela 3 apresenta os tempos de execução para as consultas da Figura 20. A abordagem SQLtoKeyNoSQLmostra-se consideravelmente superior em termos de desempenho. Para $C1$, o tempo de execução do SimpleSQL é $3x$ maior que o tempo gasto pela abordagem proposta. A diferença nos tempos de acesso reside na forma em que os dados são acessados. O SimpleSQL acessa um registro de cada vez no SimpleBD, enquanto que, graças ao método *GetN* a SQLtoKeyNoSQL pode acessar até 2500 (limite imposto pelo SimpleDB) registros por acesso ao BD. O algoritmo de junção implementado pela SQLtoKeyNoSQL (*Hash Join* clássico) também mostrou-se mais eficiente se

comparado ao utilizado pelo SimpleSQL (combinação de chaves por similaridade). Mais detalhes sobre o algoritmo de junção utilizado pelo SimpleSQL são fornecidas no próximo capítulo.

4.2.2 Experimento Unity

Unity é uma camada que possibilita o acesso através de SQL a dados armazenados nos BDs NoSQL MongoDB ou Cassandra. Além disso, o Unity possibilita o acesso simultâneo a BDR (não necessariamente na nuvem) e BDs NoSQL. Assim, é possível que tabelas de um BDR estejam armazenadas tanto em um BDR quanto em um BD NoSQL. O Unity também oferece suporte a junções, que podem ser executadas a partir de tabelas mantidas em diferentes BDs alvo.

Este experimento visa comparar o desempenho de instruções de consulta do Unity com a SQLtoKeyNoSQL. Ele realiza a execução de 4 consultas, sendo 2 delas consultas simples sem filtros e outras 2 consultas que definem junções. Neste experimento, não são utilizados filtros nas consultas tendo em vista que a versão de avaliação do Unity não provê esta capacidade. A Figura 21 apresenta as consultas que foram executadas.

```
C1.:  SELECT nu_candidato, cd_raca FROM candidato;

C2.: SELECT * FROM opcaoCandidato;

C3.: SELECT
        candidato.nu_candidato, candidato.cd_raca,
candidato.cd_sexo, evento.nu_ano
       FROM  candidato INNER JOIN evento
         ON candidato.cd_evento = evento.cd_evento;

C4.: SELECT
        candidato.nu_candidato, candidato.cd_sexo,
opcaoCandidato.cd_opcao, evento.nu_ano
       FROM  candidato INNER JOIN evento
         ON candidato.cd_evento = evento.cd_evento
       JOIN opcaoCandidato
         ON candidato.nu_candidato = opcaoCandidato.nu_candidato;
```

Figura 21 – Consultas consideradas no experimento do Unity.

Com base na execução destas consultas foi gerado o gráfico apresentado pela Figura 22. Percebe-se, pelo gráfico, que a abordagem proposta nesta dissertação possui um desempenho superior ao Unity para as 2 primeiras consultas. Este resultado indica que a forma de acesso adotada pela abordagem é promissora e é vantajosa sobre a técnica utilizada pelo Unity. Porém, o Unity apresentou melhor desempenho no processamento de junções. Um possível motivo é que o algoritmo de junção utilizado pelo Unity é uma variação do

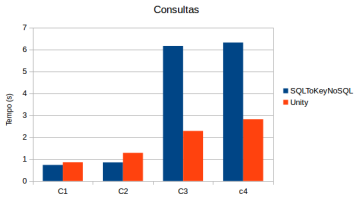

Figura 22 – Comparação do tempo de execução entre Unity e SQLtoKeyNoSQLpara consultas.

Hash Join que realiza otimizações na combinação e no acesso aos registros, enquanto que o algoritmo utilizado pela SQLtoKeyNoSQL foi o *Hash Join* clássico sem otimizações.

Apesar da SQLtoKeyNoSQLter apresentado um desempenho inferior ao Unity na execução de junções, a abordagem é flexível e permite que o algoritmo de junção possa ser modificado. Esta flexibilidade permite que novos algoritmos de junção possam ser projetados e futuramente avaliados visando melhorar o desempenho da abordagem.

4.3 CONSIDERAÇÕES FINAIS

Este capítulo apresentada algumas avaliações experimentais visando verificar a eficácia da abordagem proposta nesta dissertação. Inicialmente, experimentos para avaliação do *overhead* apresentado pela camada foram executados. Através destes experimentos pode-se afirmar que a camada possui um *overhead* não proibitivo.

Além disso, experimentos de comparação entre a abordagem proposta e outras 2 abordagens da literatura (SimpleSQL e Unity) foram realizados. SQLtoKeyNoSQL demonstrou um desempenho superior na execução das instruções SQL comparado com o SimpleDB. Em particular, apresentou um desempenho significativo na execução de consultas com filtros e no processamento de junções. Já com relação ao Unity, a abordagem apresentou um desempenho levemente superior na execução de consultas simples e um desempenho inferior na execução de junções. Estes resultados são interessantes no sentido de ressaltar a necessidade de aprimoramento, principalmente dos algoritmos de junção, na abordagem proposta.

5 TRABALHOS RELACIONADOS

Este capítulo descreve as abordagens relacionadas ao foco desta dissertação presentes na literatura. Elas são classificadas neste trabalho em duas categorias: *Layer* e *Storage Engine*. Abordagens do tipo *Layer* implementam uma camada de software que define uma abstração sobre um BDs NoSQL, permitindo ao usuário a definição e a manipulação dos dados utilizando instruções SQL. Já abordagens do tipo *Storage Engine* modificam o mecanismo de armazenamento de BDRs. Desta forma, os dados são criados e manipulados em um ambiente relacional, porém, armazenados em um BDs NoSQL. Para cada uma das abordagens são descritos o modelo de dados NoSQL utilizado, as regras de mapeamento propostas entre os modelos e as estratégias para processamento de junções.

Todos os exemplos de mapeamento apresentados são baseados em um esquema relacional para um domínio de *Cinema* constituído pelas tabelas da Figura 23 (estas tabelas já foram apresentadas na Figura 5(A)).

5.1 ABORDAGENS DO TIPO LAYER

Abordagens classificadas como *Layer* traduzem instruções SQL para as interfaces de acesso específicos dos BDs NoSQL. Uma camada é implementada entre as requisições do usuário e a interface de acesso ao BD NoSQL. Ela é responsável por receber as instruções SQL, traduzí-las e executá-las sobre o BD NoSQL. Quatro propostas são descritas nesta seção: *SimpleSQL*, que implementa uma camada sobre o SimpleDB (FERREIRA; CALIL; MELLO, 2013); *JackHare*, que é uma camada sobre o HBase (CHUNG et al., 2013); *Unity*, uma camada que permite acesso a dados armazenados em BDRs e BDs NoSQL na nuvem, mais especificamente, o BD NoSQL MongoDB e o Cassandra(LAWRENCE, 2014); e *Rith et al.*, que permite

Tabela Filmes			
id	nome	diretor	ano
1	Psyco	1	1960
11	The Godfather	2	1972
111	Patton	2	1970

Tabela Diretores		
id	nome	premios
1	Alfred Hitchcok	31
2	Francis Coppola	50

Figura 23 – BD Relacional no Domínio de Cinema.

consulta a dados relacionais e NoSQL, em particular, os BDs MongoDB e Cassandra (RITH; LEHMAYR; MEYER-WEGENER, 2014). As subseções a seguir descrevem algumas características dessas abordagens e, ao final, elas são comparadas com a solução proposta nesta dissertação.

5.1.1 Estratégia de Mapeamento

SimpleSQL é uma abordagem que permite que usuários executem instruções SQL DDL e DML sobre o SimpleDB, que implementa o modelo de dados orientado a documento. O modelo de dados do SimpleDB possui os conceitos de *domínio*, *item*, *atributo* e *valor*. Um domínio é composto por um nome *dom* e um conjunto de itens it_i na forma $(dom,\{it_1,\ldots,it_n\})$. Cada item é composto de um nome *name* e uma coleção m de atributos na forma $\{name: \{key_1 : value_1;\ldots; key_m : value_m\}\}$, sendo cada key_i o nome de um atributo (sua chave de acesso) e $value_i$ o valor de key_i. O SimpleSQL converte um BDR *db* em um domínio com nome *db*. Cada um dos itens deste domínio representa um chave primária *pk* (o nome do item, *i.e.*, *name*) de uma tabela *t* de *db*, ou seja, cada linha de *t* é representada por um item do SimpleDB. Para cada item, o SimpleSQL cria um atributo especial chamado *SimpleSQL_TableName* que armazena o nome de *t*. Por exemplo, a tabela *Filmes* é armazenada no domínio *Cinema* como: $\{1 : \{SimpleSql\_TableName$: "$filmes$"; *nome* : "*Psycho*";$diretor$: "1"; *ano* : "1960"$\}\}$, $\{11 : \{SimpleSQL\_TableName$: "$filmes$"; *nome* : "$TheGodfather$"; $diretor$: "2"; *ano* : "1972"$\}\}$ e $\{111 : \{SimpleSQL\_TableName$: "$filmes$"; *nome* : "*Patton*"; $diretor$: "2"; *ano* : "1970"$\}\}$.

JackHare é um *framework* composto por um compilador SQL, um driver JDBC e um método que utiliza a tecnologia *Map-Reduce* (DEAN; GHEMAWAT, 2008) para acessar dados armazenados no HBase. HBase é um BD NoSQL que segue o modelo de dados orientado a colunas. Um BD neste modelo é composto por um *namespace*. Cada *namespace* possui uma série de *HTables* que, por sua vez, possuem um conjunto de *famílias de colunas*. Cada família de coluna possui um conjunto de colunas acessadas por chaves e compostas por colunas e valores. Não é requerido que todas as ocorrências de uma HTable tenham as mesmas colunas.

O mapeamento realizado pelo JackHare segue as seguintes regras: (i) cada BDR *db* é mapeado para uma HTable *K* com o mesmo nome de *db*; (ii) cada tabela *t* pertencente à *db* é mapeada para uma família de colunas *f* com o mesmo nome de *t*; (iii) cada linha *r* de *t* é mapeada para um conjunto de pares chave-valor *K* que representa as suas colunas e respectivos valores.

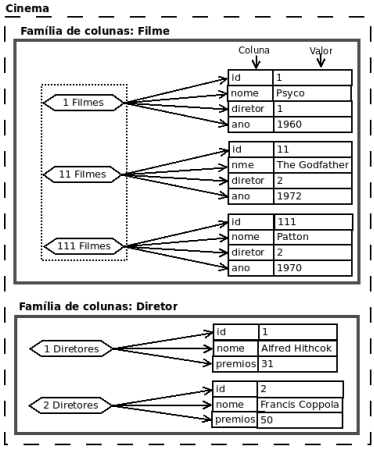

Figura 24 – Tabelas do BD Cinema mapeadas para o HBase através do JackHare.

A chave deste conjunto é obtida concatenando a chave primária de r com o nome de t; (iv) uma coluna c de r é mapeada para uma coluna C pertencente à K.

A Figura 24 ilustra o resultado do mapeamento do BD Cinema para o HBase. A borda mais externa representa um BD HBase. A Figura apresenta também as famílias de colunas *Filmes* e *Diretores*. As tuplas da tabela *Filmes*, por exemplo, estão representadas na família de colunas *Filmes*. Os hexágonos apresentam os identificadores do conjunto de colunas e as setas apontam para suas respectivas colunas e valores.

Outra abordagem que implementa uma camada de acesso relacional a BDs NoSQL é o *Unity*. Esta abordagem utiliza o BD NoSQL MongoDB para manter os seus dados. MongoDB também é um BD orientado a documento, sendo seu modelo de dados composto pelos conceitos de *BD*, *coleções de documentos* e *documentos*. Documentos são compostos por um conjunto de pares chave-valor. Unity realiza o mapeamento de um BDR db para um BD MongoDB M_{db} com o mesmo nome de db. Cada uma das coleções M_c de M_{db} representa uma tabela t de db. Cada linha r de t é um documento $M_d \in M_c$. Atributos e valores de r são representados por pares $nome\_atributo : valor\_atributo$. Por exemplo, a tabela Filmes é convertida para a seguinte coleção de documentos: $\{1 : \{nome : \text{“}Psycho\text{”}; diretor : \text{“}1\text{”}; ano : \text{“}1960\text{”}\}\}$, $\{11 : \{nome : \text{“}TheGodfather\text{“}; diretor : \text{“}2\text{“}; ano : \text{“}1972\text{“}\}\}$ e $\{111 : \{nome : \text{“}Patton\text{“}; diretor : \text{“}2\text{“}; ano : \text{“}1972\text{“}\}\}$.

Rith et al. apresenta uma abordagem que permite a execução de con-

sultas SQL no MongoDB e no Cassandra. Para tanto, são criados conectores que realizam o *parsing* de consultas SQL, enviam para o BD alvo e retornam o resultado. Não são fornecidos detalhes do mapeamento de instruções de consulta SQL. Mesmo assim, imagina-se que este mapeamento seja trivial para consultas envolvendo apenas uma tabela, uma vez que ambos os BDs alvo possuem uma linguagem de consulta semelhante em sintaxe ao padrão SQL.

5.1.2 Processamento de Junções

BDs NoSQL oferecem, em geral, operações que manipulam registros individuais, diferentemente dos BDR, onde as operações são baseadas em conjuntos, incluindo operações de junção. Assim sendo, enquanto o padrão SQL permite consultas complexas com junções, BDs NoSQL não suportam tal funcionalidade, cabendo à solução de interoperabilidade tratar o processamento de junções.

No caso do *SimpleSQL*, uma consulta é decomposta em uma lista de atributos, tabelas, junções e filtros. Caso existam junções, ele os divide em consultas simples com filtros, se necessário. Após obter o resultado de cada uma destas consultas, é criada uma tabela que respeita o esquema do resultado esperado. Os itens de resposta do resultado são combinados por similaridade utilizando as chaves primárias e estrangeiras do esquema relacional, tomando por base as informações de mapeamento entre os esquemas relacional e de documento do *S*impleDB. O desempenho do processamento das junções é, obviamente, dependente do volume de dados envolvido.

JackHare permite que usuários executem consultas envolvendo junções utilizando-se de operações *Map-Reduce*. Inicialmente, é realizada uma verificação do tamanho das tabelas. A menor das tabelas é transformada em uma lista de chaves na fase de *M*ap. A lista de chaves é armazenada no HBase. Na sequência, combinam-se os valores da tabela maior com a lista de chaves, atualizando a lista. Este processo também é feito com operações *Map-Reduce* a fim de se valer do paralelismo para obter ganho de desempenho. Após todas as entradas terem sido combinadas, a abordagem retorna a lista de chaves já combinadas, que constitui a fase de *Reduce*. Esta técnica permite que o JackHare manipule grandes volumes de dados.

Unity possibilita o acesso integrado a diversas fontes de dados na nuvem, sejam relacionais ou BDs MongoDB. Assim sendo, se a junção for executada sobre uma única fonte relacional, a junção é processada normalmente. Caso a fonte seja um BD NoSQL, a abordagem implementa um *hash join*, com base nas chaves dos registros, para executar a operação. Caso os dados

alvo estejam armazenados em 2 fontes, o processador toma uma decisão baseado no tamanho das coleções de dados. Se ambas forem grandes, os dados são extraídos em paralelo e o *Unity* executa também um *hash join*. Se os dados de uma fonte forem substancialmente maiores que a outra, ele extrai os dados da menor e os utiliza como filtro na outra fonte, ou ainda copia os dados para uma tabela temporária em um BDR e então executa a junção.

A abordagem proposta por *Rith et al.* não suporta consultas com junções até o momento da redação desta dissertação.

5.2 ABORDAGENS DO TIPO STORAGE ENGINE

Abordagens nesta categoria modificam a forma pela qual o SGBD armazena seus dados, permitindo o armazenamento em um formato NoSQL. Neste caso, a camada física do SGBD relacional é alterada. Abordagens nesta categoria devem ser capazes de realizar 4 operações sobre a camada de armazenamento NoSQL: criar, ler, atualizar e deletar dados. Assim sendo, o SGBD realiza requisições para o mecanismo de armazenamento modificado, este mecanismo mapeia os dados relacionais para o modelo NoSQL e executa a operação. Nesta categoria foram encontradas 3 abordagens: *Phoenix* (ARNAUT; SCHROEDER; HARA, 2011), *CloudyStore* (EGGER, 2009) e *DQE* (VILAÇA et al., 2013).

5.2.1 Estratégia de Mapeamento

Phoenix implementa uma nova *storage engine* para o SGBD MySQL armazenando seus dados no *Scalaris*. *Scalaris* é um BD NoSQL baseado no modelo chave-valor. Neste modelo, a chave é um identificador único para um elemento (item de dado) e o valor é uma descrição do elemento. O valor pode ser de 2 tipos: simples ou complexo. Um valor simples é um valor atômico (*e.g.*, *string* e *integer*). Já um valor complexo é constituído por um conjunto de pares chave-valor.

Phoenix define um modelo de dados intermediário para realizar o mapeamento relacional-chave-valor, chamado VOEM ($Value-based\ OEM$). VOEM é uma extensão do modelo OEM ($Object\ Exchange\ Model$) (PAPAKONSTANTINOU; GARCIA-MOLINA; WIDOM, 1995). OEM é um modelo de descrição de objetos, sendo que cada objeto instanciado possui um identificador único (*oid*). VOEM estende as capacidades do OEM com a noção de chave, ou seja, um subconjunto de atributos do objeto que identifica um objeto através de valores. Desta forma, é possível identificar uma tupla.

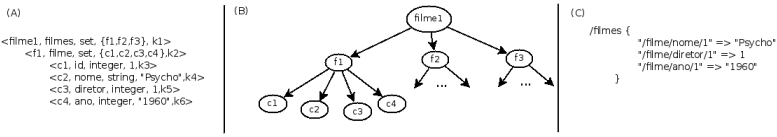

Figura 25 – Mapeamento da Tabela Filmes para o Phoenix.

Um objeto VOEM é descrito pela sêxtupla v=$< oid, \lambda, \tau, v, k >$, onde o *oid* é o identificador do objeto, λ é o rótulo do objeto, τ indica o tipo do objeto (simples ou complexo), e v é o valor do objeto. Se o tipo do objeto for complexo, então v contém um conjunto de *oid*. A última parte de um objeto VOEM k é uma chave que identifica unicamente um par chave-valor. A Figura 25(A) apresenta a codificação VOEM da tabela $Filmes$.

VOEM permite o mapeamento do modelo relacional para o modelo chave-valor. Este mapeamento é baseado nas seguintes regras: (i) uma linha r de uma tabela relacional t é representada por um objeto VOEM O_i com rótulo t e seu valor é uma coleção de *oids* que mapeiam todas as colunas de r; (ii) cada coluna c de tipo τ de r é mapeada para um objeto VOEM O_i com rótulo c e tipo τ.

A coleção de objetos é transformada em um grafo VOEM rotulado V (Figura 25(B)). Cada vértice de V é um objeto e cada uma das arestas um relacionamento entre objetos. Este grafo é usado para criar a chave de cada objeto VOEM O. Para definir a chave, é feita uma busca em profundidade cuja origem é o primeiro nodo de V e destino o vértice O. Na Figura 25(C), o par chave-valor "$/filme/nome/1 => Psyco$", tem sua chave definida através da concatenação dos rótulos do caminho do objetos VOEM $f1$ ($filme$), $c2$ ($nome$) (Figura 25(B)) e do valor 1 da chave primária (representada no grafo como $c1$) que identifica a linha mapeada.

Por fim, o mapeamento de VOEM para o modelo chave-valor é mais intuitivo: um objeto VOEM O é mapeado para um par chave-valor kv, onde a chave é a chave de O. Se o valor de O for simples, então, o valor de kv será o mesmo de O. Se o valor for complexo, o valor de kv será o conjunto de valores dos objetos VOEM que formam o valor de O mapeados para o modelo chave-valor. Por exemplo, o objeto "f1"(Figura 25(A)) é apresentado já mapeado no modelo chave-valor na Figura 25(C).

CloudyStore é uma abordagem similar à *Phoenix*, porém, realiza o mapeamento para um BD NoSQL orientado a colunas denominado *Cloudy* (EGGER, 2009). O modelo de dados do Cloudy é similar ao do Hbase, ou seja, ele é composto por *keyspaces*, família de colunas, conjuntos de colunas acessados por chaves, colunas e valores.

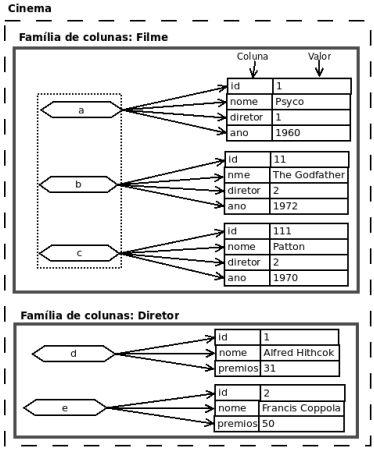

Figura 26 – BD Cinema mapeado para o HBase pelo CloudyStore.

CloudyStore propõe 3 regras para mapear um BDR para o modelo de dados do Cloudy: (i) um BD *db* é mapeado para uma *keyspace K* com o nome *db*; (ii) uma tabela *t* de *db* é mapeada para uma família de colunas com o mesmo nome de *t*; (iii) uma linha *r* de *t* é mapeada para um conjunto de pares chave-valor *rk* de *K*, sendo a chave de *rk* uma chave interna do MySQL para *r* (conhecida como *rowid*). Cada coluna *c* de *r* é mapeada para uma coluna *ck* pertencente à *rk*, cujo nome será *c*.

A Figura 26 apresenta o mapeamento do BD *Cinema*. As tabelas *Filmes* e *Diretores* são mapeadas para as famílias de colunas *Filmes* e *Diretores*, respectivamente. Cada linha de uma tabela é mapeada para pares *chave-valor*. Por exemplo, a primeira linha de *Filmes* é mapeada para um valor $\{$"`nome`" : "`Psyco`","`diretor`" : "`1`","`ano`" : "`1960`"$\}$ na família de coluna *Filmes*. As chaves *a*, *b*, *c*, *d* e *e* para as colunas das famílias (borda pontilhada na Figura 26) representam os *rowids* que identificam as tuplas no MySQL.

DQE é uma abordagem que define uma storage engine para o SGBDR Derby, armazenando os dados no BD orientado a colunas HBase. Seu mapeamento é baseado em 4 regras: (i) um BD *db* é mapeado para uma keyspace *K* com o nome de *db*; (ii) cada tabela *t* de *db* é mapeada para uma HTable *ht* nomeada com *t*; (iii) uma linha *r* de *t* é mapeada para um conjunto de pares chave-valor *rk* de *ht*; (iv) cada coluna *c* de *r* é mapeada para uma coluna *ck* pertencente à *rk*, com o nome de *c*. Se o BD possuir atributos indexados, DQE cria, para cada atributo, uma nova HTable, sendo os valores do atributo

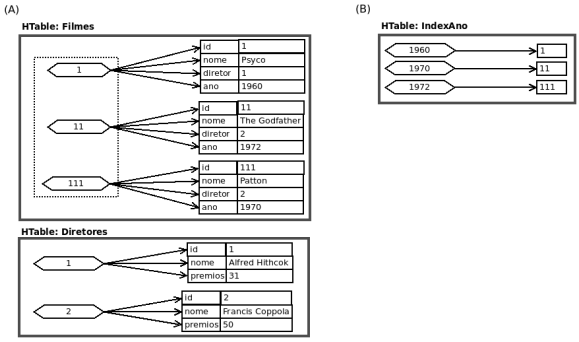

Figura 27 – BD Cinema mapeado pela Abordagem DQE.

mapeados para uma linha nesta tabela. A chave da linha é uma coluna que mantém o valor indexado e uma outra coluna atua como identificador da *rk*, indicando a linha da tabela à qual o valor indexado pertence.

A Figura 27 apresenta o mapeamento para o BD *Cinema*. O mapeamento gera 3 HTables: as duas primeiras (Figura 27(A)) correspondem às tabelas *Filmes* e *Diretores*, enquanto a terceira (Figura 27(B)) representa o mapeamento de um índice criado para o atributo *ano* da tabela *Filmes*. Nesta tabela de índice, a chave da linha é o atributo indexado (*ano*) e o valor é a chave da coluna que possui este valor de ano na respectiva HTable. Se o atributo indexado não possuir um valor único, então uma HTable diferente é criada para o atributo indexado, onde o atributo indexado será a chave e, para cada ocorrência do atributo indexado, é criado um par (coluna, valor), onde a coluna é a chave do valor na respectiva HTable e o valor é vazio.

5.2.2 Processamento de Junções

As abordagens do tipo *Storage Engines* não implementam estratégias específicas para o processamento de junções. Elas utilizam as técnicas implementadas pelos SGBDRs. O SGBDR cria o plano de consulta e o otimiza, verifica as junções a serem executadas e somente faz requisições às *storage engines* para buscar os dados envolvidos na consulta. Desta forma, o SGBDR é o responsável por este processamento.

5.3 ANÁLISE COMPARATIVA

Esta seção apresenta um compartivo entre as abordagens descritas neste capítulo. Este comparativo inclui a abordagem proposta nesta dissertação e considera as seguintes características (Tabela 4): (i) o tipo da abordagem (*Layer* ou *Storage Engine*); (ii) o SGBD NoSQL para o qual foi desenvolvido; (iii) o modelo de dados NoSQL que a abordagem interopera; (iv) quais instruções SQL a abordagem suporta (as abordagens JackHare e *Rith et al.* não implementam todos as instruções DML e por isso são marcadas como DML Restrita); (v) se a abordagem utiliza um dicionário de dados para manter informações de mapeamento; e (vi) como a abordagem processa junções, se for o caso.

Abordagem	Categoria	BD NoSQL	Modelo NoSQL	Suporte SQL	Dicionário	Junções
SimpleSQL (2013)	Layer	SimpleBD	Documento	DDL+DML Subset	Sim	Similaridade
JackHare (2013)	Layer	HBase	Colunar	DML Restrita + DDL	Sim	Map-Reduce
Unity (2014)	Layer	MongoDB/Cassandra	Documento/Colunar	DML Subset	Sim	Hash-Join
Rith et al. (2014)	Layer	Cassandra/MongoDB	Colunar/Documento	DML Subset	-	-
Phoenix (2011)	Storage Engine	Scalaris	Chave-valor	Instruções MySQL	Não	SGBD
CloudyStore (2009)	Storage Engine	Cloudy	Colunar	Instruções MySQL	Sim	SGBD
DQE (2013)	Storage Engine	HBase	Colunar	DDL+DML Subset	Sim	SGBD
SQLtoKeyNoSQL(2015)	Layer	Baseados em Chave	Baseados em Chave	DDL+DML Subset	Sim	Hash-Join

Tabela 4 – Comparativo dos Trabalhos Relacionados.

Cada uma das abordagens possui um forma única para realizar o mapeamento do esquema de dados e das instruções SQL. Nota-se que existem soluções para diversos modelos de dados NoSQL, porém, cada uma das abordagens realiza o mapeamento de maneira diferente. Mesmo as abordagens JackHare e DQE, que realizam o mapeamento para o mesmo BD NoSQL alvo (Hbase) possuem mapeamentos distintos para alguns conceitos, tornando o mapeamento algo muito singular. O mapeamento executado pela SQLtoKeyNoSQL é apresentado na seção 3.2 e difere-se das demais abordagens por não depender unicamente de um BD NoSQL específico, mas sim dos modelos de dados, apresentando um nível de abstração maior e possibilitando maior flexibilidade.

Ainda, a grande maioria das abordagens é limitada ao mapeamento de apenas um modelo de dados NoSQL. Apesar da abordagem Unity suportar mais de um modelo de dados alvo, sua arquitetura é baseada em conectores que são responsáveis pelo mapeamentos dos modelos de dados e das instruções. Este conectores são complexos e devem ser desenvolvidos pelo usuário. A abordagem proposta nesta dissertação contribui com uma abstração (modelo canônico) para a realização do mapeamento. O modelo canônico proposto permite que um esquema relacional possa ser mapeado para qualquer um dos BDs NoSQL baseados em chave, independente do mo-

delo de dados utilizado. O usuário também é responsável pelo desenvolvimento de conectores, porém os conectores apenas devem ser implementar as 4 operações apresentadas na seção anterior.

Outro ponto a salientar é que as abordagens baseadas em *Storage Engine*, por estarem implantadas em um SGBDR específico, suportam, em geral, o mapeamento de qualquer operação SQL disponibilizada por este SGBDR. Já as abordagens baseadas em *Layer* limitam-se a mapear apenas algumas instruções SQL DML e/ou DDL, devido ao custo de implementar a transformação de cada uma delas para o SGBD NoSQL alvo. Em contrapartida, abordagens do tipo *Layer* são mais flexíveis, uma vez que não são atreladas a um SGBD relacional específico.

A presença de um dicionário é fundamental para as abordagens baseadas em *Layer*, pois a gerência do mapeamento é feito por uma camada de software específica para a interoperabilidade e tal informação é vital para proceder as conversões estruturais e de instruções. Mesmo assim, verifica-se que boa parte das abordagens *Storage Engine* também o consideram.

Um desafio na efetivação da interoperabilidade entre o modelo relacional e os modelos NoSQL é o suporte a junções, inexistente em BDs NoSQL para não comprometer a disponibilidade e a escalabilidade de acesso, e que deve ser provido pela abordagem. Conforme indicado na Tabela 4, abordagens da categoria *Storage Engine* utilizam o SGBDR para o processamento de junções. Já as abordagens baseadas em *Layer* utilizam diversas técnicas, não havendo um consenso, conforme descrito a seguir.

O *SimpleSQL* utiliza uma técnica de junção baseada em similaridade, combinado os dados utilizando as chaves primárias e estrangeiras do esquema relacional. O *JackHare* executa junções utilizando *Map-Reduce*, combinando os valores pelas respectivas chaves através de paralelismo. *Unity* executa junções utilizando uma técnica de *hash join* onde as chaves hash são o conjunto de chaves primarias da linha. A abordagem SQLtoKeyNoSQL também é capaz de processar junções. Ela suporta o *Hash Join* (onde a chave *hash* são os atributos da condição de junção) e o *Sort-Merge Join* (as linhas são ordenadas com base nos atributos da condição de junção) e sua arquitetura permite que o algoritmo de junção seja alterado e novos sejam implementados. A abordagem proposta por *Rith et al.* não é capaz de processar junções.

Na literatura também foram encontrados outros trabalhos semelhantes, porém, com um foco diferente do adotado nesta dissertação. O *Yesquel* (AGUILERA et al., 2015) é uma gerenciador de dados que possui suporte SQL com características apresentadas por BDs NoSQL (disponibilidade, escalonamento, etc). Entretanto, diferentemente da SQLtoKeyNoSQL, o *Yesquel* propõe um nova arquitetura de BD com estas capacidades. Outro foco é o adotado pelo *MoSQL* (TOMIC; SCIASCIA; PEDONE, 2013), que

propõe uma *Storage Engine* para o MySQL mantendo os dados armazenados de maneira distribuída, porém em memória, a fim de proporcionar maior disponibilidade e escalabilidade horizontal aos dados. Outra abordagem relacionada é o *SOS-Plataform* (ATZENI; BUGIOTTI; ROSSI, 2012), que desenvolve uma camada para padronizar o acesso a dados em BDs NoSQL. Entretanto, ela não oferece suporte para instruções SQL nem processamento de junções. Ela define apenas um mapeamento baseado em métodos da API REST, diferente da abordagem proposta nesta dissertação, que define um modelo canônico como abstração dos modelos de dados NoSQL.

Com base nesta análise da literatura é possível afirmar que a abordagem SQLtoKeyNoSQLapresenta, como principal contribuição, a sua flexibilidade de mapeamento, que possibilita o armazenamento de dados relacionais e o seu acesso, de forma transparente, a diversos BDs NoSQL com modelos de dados baseados em chave de acesso.

6 CONCLUSÃO

A demanda por gerenciamento de *Big Data* vem crescendo nos últimos anos. Estas demandas incluem alta disponibilidade, alto desempenho no tratamento dos dados e alta escalabilidade. BDRs não são totalmente adequados a aplicações que requerem este tipo de demanda. Para atender essas necessidades, novas soluções de gerenciamento de dados, os denominados BDs NoSQL, estão sendo cada vez mais utilizados. Os modelos de dados implementados pelos BDs NoSQL não são compatíveis com o modelo de dados relacional e, como consequência, não suportam o padrão SQL de consulta. Este é um impedimento para um grande número de aplicações que utilizam SGBDRs e não desejam arcar com um alto custo de migração e curva de aprendizagem para tecnologias NoSQL.

Assim, faz-se necessária a criação de soluções que lidem com a interoperabilidade no sentido relacional→NoSQL através da execução de mapeamentos entre modelos de dados e suas operações, e que não comprometam demasiadamente o desempenho das aplicações. Duas principais soluções podem ser vislumbradas para lidar com este problema, conforme descrito no Capítulo 5: (i) modificar os SGBDRs de forma que estes manipulem os dados em um ambiente relacional e os armazenem em BDs NoSQL; e (ii) desenvolver uma abstração/camada capaz de manipular os dados relacionais através da SQL e os armazenar em BDs NoSQL. Esta dissertação adota a segunda solução considerando a flexibilidade que ela proporciona, ou seja, independência de SGBD, gerenciamento simultâneo de múltiplas fontes de dados, dentre outras.

SQLtoKeyNoSQL é uma camada proposta nesta dissertação que permite o armazenamento e a manipulação de dados utilizando instruções SQL sobre BDs NoSQL. Para prover este acesso, a abordagem conta com um modelo canônico. O modelo é composto por uma série de chaves organizadas hierarquicamente e tem como objetivo abstrair os modelos de dados NoSQL baseados em chave de acesso, permitindo um mapeamento simples de esquemas de dados relacionais para qualquer BD NoSQL baseado em chave. Além disso, para a execução das instruções SQL sobre os BDs NoSQL, a camada provê um mapeamento de instruções para um conjunto de métodos baseados na *API REST*. Para executar tais métodos sobre os BDs NoSQL alvo, é necessário a implementação de conectores. Os conectores apenas necessitam implementar os métodos primitivos, realizando a tradução de cada um deles para a API do BD NoSQL alvo.

A solução proposta nesta dissertação tem como contribuição o modelo canônico hierárquico, uma vez que este modelo viabiliza uma abordagem que

independe do BD NoSQL alvo e depende apenas do suporte a algum modelo de dados baseado em chave. Outra contribuição desta dissertação é o conjunto de regras de mapeamento de esquemas e de instruções SQL. Estes mapeamentos possibilitam o mapeamento do esquema relacional para o modelo canônico e do modelo canônico para cada um dos BDs alvo, além do mapeamento das instruções SQL para a API de acesso de BDs NoSQL. Além disso, a arquitetura proposta para a SQLtoKeyNoSQL possui como característica principal a independência entre seus componentes, possibilitando, assim, que os componentes sejam facilmente alterados. Os experimentos realizados para avaliação da eficácia da abordagem proposta também são considerados como contribuição desta dissertação.

Os experimentos realizados possuíam como objetivo verificar o *overhead* praticado pela abordagem e a comparação com abordagens presentes na literatura. Os experimentos relacionados à *overhead* demostraram que a abordagem apresenta um *overhead* não proibitivo se comparado com o tempo de execução da execução direta das instruções no BD NoSQL. Estes experimentos também mostraram que o *overhead* da execução sobre BDs NoSQL chave-valor é maior que os demais. Esse fato se deve à simplicidade imposta por este modelo de dados. Mesmo assim, o tempo de processamento de consultas sobre os BDs do modelo chave-valor é semelhante aos demais modelos. Já os experimentos de comparação com as abordagens SimpleSQL e Unity demostraram que a SQLtoKeyNoSQL possui desempenho satisfatório e competitivo.

O experimento que realizou a comparação da abordagem proposta com o SimpleSQL demonstrou que para todas as instruções SQL executadas a SQLtoKeyNoSQL possui um desempenho superior. Destaca-se que a execução dos comandos DML na camada proposta obtiveram tempos de execução significativamente menores que os tempos apresentados pelo SimpleSQL. Este fato deve-se, principalmente, pela forma na qual os dados são acessados. Enquanto o SimpleSQL acessa todos os registros de maneira individual utilizando filtros, a abordagem proposta acessa os dados em blocos e apenas realiza filtros través da camada. Em comparação com o Unity, a abordagem proposta apresenta desempenho superior no acesso ao dados. Porém, a SQLtoKeyNoSQL possui um desempenho inferior na execução de junções. Esta diferença no desempenho do processamento das junções é devido ao algoritmo utilizado para realizar as junções. O algoritmo utilizado pelo Unity é um *Hash Join* otimizado, enquanto o algoritmo *Hash Join* implementado até o momento para a SQLtoKeyNoSQL é uma versão clássica do mesmo algoritmo sem nenhum tipo de otimização.

Esta dissertação, mesmo apresentando diferenciais em relação à literatura relacionada, apresenta alguns pontos a melhorar. Um deles é o não

particionamento das tabelas. A SQLtoKeyNoSQL armazena tabelas em diferentes BDs NoSQL alvo, porém, cada tabela deve ser armazenada em apenas um BD alvo a fim de reduzir a complexidade de gerenciamento. O particionamento de tabelas pode trazer ganhos de desempenho ao explorar melhor o paralelismo no acesso aos dados. Outro ponto a melhorar neste trabalho, é o não aproveitamento das linguagens de consulta utilizadas como forma de acesso por alguns BDs NoSQL. BDs NoSQL cujo modelo de dados é orientado a documentos ou a colunas geralmente oferecem linguagens de consulta, se comparados com os BDs chave-valor. Entretanto, justamente pelo fato dos BDs chave-valor não possuírem linguagens de consulta, mas se enquadrarem como BDs NoSQL baseados em chave de acesso, ou seja, uma das categorias de BD NoSQL alvo dessa dissertação para fins de mapeamento, a abordagem optou por não considerar o mapeamento para instruções destas linguagens de consulta. Optou-se pelo suporte apenas à capacidades básicas de consulta (interfaces de acesso) que são comuns a todos os BDs NoSQL baseados em chave de acesso. Outra limitação a ser ressaltada, é a rigidez imposta pela abordagem quanto ao formato dos esquemas BDs NoSQL utilizados, pois o esquema dos dados deve ser criado e gerenciado pela camada. Não é possível, por exemplo, a inclusão de um BD NoSQL previamente definido e o acesso aos seus dados através da camada proposta. Além disso, a abordagem proposta limita-se a utilizar o dicionário de dados em memória, apesar de armazenar uma cópia em disco.

Apesar dos pontos que necessitam de melhorias, pode-se afirmar que esta dissertação cumpre o objetivo de definir uma nova abordagem para execução de instruções SQL sobre BDs NoSQL. A abordagem SQLtoKeyNoSQL é flexível e permite que diferentes fontes de dados NoSQL sejam manipuladas simultaneamente. Ela permite também que tabelas de um mesmo BDR sejam armazenadas em diferentes BDs NoSQL simultaneamente.

6.1 TRABALHOS FUTUROS

A abordagem proposta nesta dissertação contribui com uma nova estratégia de acesso a dados armazenados em BDs NoSQL baseados em chave através da linguagem SQL. Apesar das contribuições já alcançadas com esta versão da abordagem, algumas oportunidades de melhoria podem ser consideradas. Algumas possibilidades de trabalhos futuros são as seguintes:

- Extensão do modelo canônico para que seja possível a abstração do modelo de dados NoSQL orientado a grafos;

- Experimentos que comparem a abordagem proposta com abordagens do tipo *Storage Engine*;
- Estudo comparativo para definir o melhor algoritmo de junção para cada tipo de modelo de dados, bem como qual o algoritmo que melhor se adapta a junções entre tabelas armazenadas em modelos de dados distintos;
- Suporte a índices para otimizar o acesso aos dados;
- Extensão da proposta para suportar concorrência de acesso por múltiplos usuários;
- Suporte a transações;

6.2 PRODUÇÃO DE ARTIGOS

No decorrer do desenvolvimento desta dissertação foram desenvolvidos quatro artigos. O primeiro artigo desenvolvido foi um *survey* com as abordagens de interoperabilidade (Relacional-NoSQL) encontradas na literatura. Este artigo foi publicado na ERBD de 2015 e agraciado com o prêmio de melhor artigo na trilha de pesquisa. Outro artigo, com a proposta da dissertação, foi publicado no WTDBD (Workshp de Teses e Dissertações) do SBBD 2015. O terceiro artigo foi desenvolvido com a abordagem proposta (sem o processamento de junções) e com os experimentos de *overhead*, sendo este publicado no iiWas 2015. Outro artigo, do tipo de *survey*, foi produzido para o SIGMOD RECORD, porém ainda não foi recebida resposta para esta submissão. Está previsto o desenvolvimento de mais um artigo, esse com experimentos de junções e comparações com outras abordagens encontradas na literatura.

REFERÊNCIAS

ABADI, D. J. Data management in the cloud: Limitations and opportunities. **IEEE Data Eng. Bull.**, v. 32, n. 1, p. 3–12, 2009.

ABOUZEID, A. et al. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. **Proc. VLDB Endow.**, VLDB Endowment, v. 2, n. 1, p. 922–933, ago. 2009. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/1687627.1687731>.

AGUILERA, M. K. et al. Yesquel: Scalable sql storage for web applications. In: **Proceedings of the 2015 International Conference on Distributed Computing and Networking**. New York, NY, USA: ACM, 2015. (ICDCN '15), p. 40:1–40:4. ISBN 978-1-4503-2928-6. Disponível em: <http://doi.acm.org/10.1145/2684464.2684504>.

ARNAUT, D.; SCHROEDER, R.; HARA, C. Phoenix: A relational storage component for the cloud. In: **Cloud Computing (CLOUD), 2011 IEEE International Conference on**. [S.l.: s.n.], 2011. p. 684–691. ISSN 2159-6182.

ATZENI, P.; BUGIOTTI, F.; ROSSI, L. Uniform access to non-relational database systems: The sos platform. In: **Advanced Information Systems Engineering**. [S.l.]: Springer Berlin Heidelberg, 2012. (Lecture Notes in Computer Science, v. 7328), p. 160–174. ISBN 978-3-642-31094-2.

BERMAN, J. J. **Principles of big data: preparing, sharing, and analyzing complex information**. [S.l.]: Newnes, 2013.

BERTINO, E.; MARTINO, L. **Object-Oriented Database Systems: Concepts and Architectures**. [S.l.]: Addison-Wesley Pub, 1993.

BOS, B.; W3C. **XML representation of a relational database**. 1997. Disponível em: <https://www.w3.org/XML/RDB.html>.

CALIL, A.; MELLO, R. dos S. SimpleSQL: A relational layer for SimpleDB. In: **Proceedings of the 16th East European Conference on Advances in Databases and Information Systems**. Berlin, Heidelberg: Springer-Verlag, 2012.

CATTELL, R. Scalable sql and nosql data stores. **SIGMOD Rec.**, ACM, New York, NY, USA, v. 39, n. 4, p. 12–27, maio 2011. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/1978915.1978919>.

CHUNG, W.-C. et al. Jackhare: a framework for SQL to NoSQL translation using MapReduce. **Automated Software Engineering**, Springer, p. 1–20, 2013.

DEAN, J.; GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In: . New York, NY, USA: [s.n.], 2008.

EGGER, D. **SQL in the Cloud**. Tese (Doutorado) — Master Thesis ETH Zurich, 2009, 2009.

FERREIRA, G. dos S.; CALIL, A.; MELLO, R. dos S. On providing DDL support for a relational layer over a document NoSQL database. In: **Proceedings of International Conference on Information Integration and Web-based Applications Services**. New York, NY, USA: ACM, 2013.

FIELDING, R. T. **Architectural styles and the design of network-based software architectures**. Tese (Doutorado) — University of California, Irvine, 2000.

HOFER, C.; KARAGIANNIS, G. Cloud computing services: taxonomy and comparison. **Journal of Internet Services and Applications**, Springer-Verlag, v. 2, n. 2, p. 81–94, 2011. ISSN 1867-4828. Disponível em: <http://dx.doi.org/10.1007/s13174-011-0027-x>.

KLEIN, J.; GORTON, I. Runtime performance challenges in big data systems. In: **Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development**. New York, NY, USA: ACM, 2015. (WOSP '15), p. 17–22. ISBN 978-1-4503-3340-5. Disponível em: <http://doi.acm.org/10.1145/2693561.2693563>.

KUMAR, R. et al. Apache hadoop, nosql and newsql solutions of big data. **International Journal of Advance Foundation and Research in Science & Engineering (IJAFRSE)**, v. 1, n. 6, p. 28–36, 2014.

LAWRENCE, R. Integration and virtualization of relational SQL and NoSQL systems including MySQL and MongoDB. In: **Computational Science and Computational Intelligence (CSCI), 2014 International Conference on**. [S.l.: s.n.], 2014.

MISHRA, P.; EICH, M. H. Join processing in relational databases. **ACM Computing Surveys (CSUR)**, ACM, v. 24, n. 1, p. 63–113, 1992.

PAPAKONSTANTINOU, Y.; GARCIA-MOLINA, H.; WIDOM, J. Object exchange across heterogeneous information sources. In: **Data Engineering, 1995. Proceedings of the Eleventh International Conference on**. [S.l.: s.n.], 1995.

RITH, J.; LEHMAYR, P. S.; MEYER-WEGENER, K. Speaking in tongues: SQL access to NoSQL systems. In: ACM. **Proceedings of the 29th Annual ACM Symposium on Applied Computing**. [S.l.], 2014.

SADALAGE, P. J.; FOWLER, M. **NoSQL distilled: a brief guide to the emerging world of polyglot persistence**. [S.l.]: Pearson Education, 2012.

STONEBRAKER, M. New opportunities for new sql. **Commun. ACM**, ACM, New York, NY, USA, v. 55, n. 11, p. 10–11, nov. 2012. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/2366316.2366319>.

TOMIC, A.; SCIASCIA, D.; PEDONE, F. Mosql: An elastic storage engine for mysql. In: **Proceedings of the 28th Annual ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2013. (SAC '13), p. 455–462. ISBN 978-1-4503-1656-9. Disponível em: <http://doi.acm.org/10.1145/2480362.2480452>.

VILAÇA, R. et al. An effective scalable SQL engine for NoSQL databases. In: SPRINGER. **Distributed Applications and Interoperable Systems**. [S.l.], 2013.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: state-of-the-art and research challenges. **Journal of Internet Services and Applications**, Springer-Verlag, v. 1, n. 1, p. 7–18, 2010. ISSN 1867-4828. Disponível em: <http://dx.doi.org/10.1007/s13174-010-0007-6>.

ZIKOPOULOS, P. C. et al. Understanding big data. **New York et al: McGraw-Hill**, 2012.