UNIVERSIDADE FEDERAL DE SANTA CATARINA

# Mapeamento de Modelagem Lógica de Dados Orientado a Agregados para Sistemas de Gerência de Bancos de Dados NoSQL

Nathan Reuter Godinho

Florianópolis 2018/1

Nathan Reuter Godinho

# Mapeamento de Modelagem Lógica de Dados Orientado a Agregados para Sistemas de Gerência de Bancos de Dados NoSQL

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do título de Bacharel, do curso de Ciência da Computação na Universidade Federal de Santa Catarina.

Orientador: Prof$^{o}$ Ronaldo dos Santos Mello

Florianópolis

2018/1

Nathan Reuter Godinho

# Mapeamento de Modelagem Lógica de Dados Orientado a Agregados para Sistemas de Gerência de Bancos de Dados NoSQL

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do título de Bacharel, do curso de Ciências da Computação na Universidade Federal de Santa Catarina.

---

**Prof$^{o}$ Rafael Cancian, Dr.**

Coordenador do Curso

**Banca Examinadora:**

---

**Prof$^{o}$ Ronaldo dos Santos Mello, Dr.**
Orientador
Universidade Federal de Santa Catarina

---

**Angelo Augusto Frozza, M.SC.**
Instituto Federal Catarinense

---

**Rodrigo Gonçalves**
Universidade Federal de Santa Catarina

Florianópolis 2018/1

# RESUMO

Com a popularização dos bancos de dados NoSQL, que se deu devido a sua alta escalabilidade, ausência de esquema previamente definido e suporte a grande volume de dados, surgiram novos obstáculos ao se desenvolver aplicações com esse tipo de banco. Esses problemas variam desde a escolha de qual tipo de banco selecionar para a devida aplicação até como criar esquemas para esse tipo de banco não necessariamente estruturados, a fim de facilitar tarefas como recuperação, integração e análise dos dados. Pensando em aproveitar conhecimentos prévios dos desenvolvedores em modelagem de dados relacional, esse trabalho propõe criar regras de mapeamento de um modelo lógico NoSQL baseado em agregados, desenvolvido como um trabalho de mestrado no grupo de banco de dados da UFSC, para modelos físicos em bancos de dados NoSQL baseados igualmente em agregados, que é o caso de bancos de dados NoSQL do tipo documento, colunar e chave-valor. A ferramenta de modelagem de banco de dados BrModeloNext é estendida para dar suporte a esta etapa de projeto de bancos de dados NoSQL através da implementação dessas regras de mapeamento.

**Palavras-Chave:** Projeto Físico, NoSQL, Modelagem, Ferramenta de Banco de dados, BrModeloNext.

# Abstract

With the popularization of the NoSQL databases, that came by the high scalability, lack of previously defined schemes and support of big volume of data, new obstacles emerged when developing aplications with this type of database. This problems range from a choosing which type of database to select for de right aplication, to creating schemes for those types of databases, which are not necessarily structured, in order to favor tasks such as retrieval, integration and data analysis. In order to take advantage of previus knowledge from the developers in relational data modeling, this work proposes to create a mapping of a NoSQL logical model based on agregates, developed as a master's work in UFSC database group to physical models in NoSQL databases that are also based on aggregates, wich is the case for document, column and key-value database. The database modeling tool BrModeloNext is extended to support this stage of NoSQL database design through the implementation of these mapping rules.

**Keywords:** Physical Project, NoSQL, Modeling, DataBase Modeling Tool, BrmodeloNext

# LISTA DE ILUSTRAÇÕES

# LISTA DE ABREVIAÇÕES E SIGLAS

| | |
|---|---|
| BD | *Banco de Dados* |
| ACID | *Atomicity, Consistency, Isolation , Durability* |
| BASE | *Basically Available, Soft state, Eventual consistency* |
| MVCC | *Multiversion Concurrency Control* |
| NoSQL | *Not Only SQL* |

# SUMÁRIO

# 1 Introdução

Há anos a forma de armazenar os dados vem mudando. Dados estão presentes em volume cada vez maior, devido a fenômenos como IoT (Internet of Things) e o barateamento do custo de se armazenar grandes volumes de dados (DEVAN, 2017). Além do volume, a velocidade de acesso tem uma taxa de crescimento superior a cada ano (DEVAN, 2017), seja para escrita ou leitura de dados e somada a variedade de diferentes dados que tem que ser tratada, define-se o desafio dos 3 v's , de volume, velocidade e variedade, do chamado *Big Data* (ALMEIDA, 2017).

O fenômeno *Big Data* foi o estopim para o surgimento de diversos bancos de dados (BD) não relacionais denominados BD NoSQL(Not Only SQL). BD NoSQL são diferentes dos BD relacionais, que se baseiam nos princípios ACID (*Atomicity, Consistency, Isolation, Durability*) (NAVATHE&ELMASRI, 2017), e não são tão indicados para aplicações que envolvem grande volume de dados, alta velocidade de acesso e uma variação grande no formato de dados. Estudos revelam que, em BD voltados para *Big Data*, trocar consistência por disponibilidade pode levar a ganhos altíssimos em escalabilidade, além de velocidade de acesso (PRITCHETT, 2008). Desta forma, em BD NoSQL os princípios mais comuns são BASE (Basically *Available, Soft state, Eventual consistency*) e MVCC (*Multiversion concurrency control*) (MIHALCEA, 2017). Esses princípios fornecem mais liberdade para os BD gerenciarem sua consistência, que pode ser totalmente consistente ou eventualmente ou ambos (programáveis) como é o caso do MongoDB (MONGODB, 2016). Essa consistência fraca permite escritas e leituras mais rápidas e assíncronas entre os nodos do BD, ao mesmo tempo que leva a uma confiabilidade menor em relação aos dados.

Outro ponto são as estruturas dos dados em BD relacionais, onde se tem uma estrutura fixa definida de antemão pelo esquema ao projetar o modelo do BD. Já nos BD NoSQL, os dados podem ser tanto estruturados quanto semiestruturados e não-estruturados e, devido a essa variedade, não ter nenhuma estrutura definida *a priori*, sendo chamados de *schemaless*.

Quando um BD é *schemaless*, armazenar dados se torna muito mais casual (SADALAGE, 2012), pois um BD NoSQL do tipo chave-valor, por exemplo, pode manter qualquer valor para uma dada chave. De mesma forma, um BD NoSQL de documento, que é um banco que apresenta estruturas como XML e JSON, faz o mesmo sem restringir o formato do documento que se está armazenando. Com um esquema, é necessário saber de antemão como vai ser a estrutura dos dados, o que em alguns casos pode ser muito difícil.

Assim o fato de BD's NoSQL possuírem diferentes implementações e serem *schemaless* leva a diferentes problemáticas ao projetar BD NoSQL, que envolve escolher as categorias adequadas de BD para cada classe de aplicação, saber manter os dados de forma a possibilitar fácil acesso e, acima de tudo, possibilitar um esquema pré-definido, de forma que se mantenha um alinhamento dos dados utilizando esse tipo de BD, que é o foco deste trabalho.

## 1.1 Motivação

Um dos motivos que desperta o interesse pelos BD NoSQL é a falta de definição de uma estrutura prévia para seus dados. Apesar disso ser bom para a maioria das aplicações em que eles são utilizados, existem casos em que se deseja associar a um BD NoSQL a um esquema predefinido.

Como grande parte dos usuários que desejam definir esquemas para BD NoSQL possuem certo nível de experiência em projeto de BD relacionais a partir de uma modelagem conceitual (modelo entidade-relacionamento, por exemplo), é desejável que se possa unir os dois casos e ter uma ferramenta que mapeie modelagens conceituais para modelos lógicos NoSQL. Esse é o objetivo da ferramenta BrModeloNext (MENNA, RAMOS e MELLO, 2011) que contém todas as funcionalidades de modelagem de dados relacionais das antigas versões da ferramenta BrModelo (CÂNDIDO, 2005) e foi estendida para suportar a modelagem lógica de BD NoSQL a partir da metodologia definida no trabalho de Cláudio Lima (LIMA, 2016). Esta metodologia define regras de conversão de uma modelagem conceitual entidade-relacionamento para uma modelagem lógica baseada em agregados.

Assim sendo, surge o propósito do presente Trabalho de Conclusão de Curso, que é desenvolver um módulo para conversão de um modelo lógico de agregados, a partir do proposto em Lima (2016), para modelagens físicas nos BD NoSQL MongoDB, Cassandra e Redis, na ferramenta BrModeloNext, a fim de suportar o processo completo de projeto de um BD para diferentes modelos de dados NoSQL.

## 1.2 Objetivos

## 1.3 Objetivo Geral

O presente Trabalho de Conclusão de Curso tem o objetivo de desenvolver um processo para projeto físico de BD NoSQL baseados em agregados, a partir de um conjunto de regras que envolvem o mapeamento de modelo lógico NoSQL para modelos de BD NoSQL baseados em agregados, incorporando este processo à ferramenta BrModeloNext.

## 1.4 Objetivos Específicos

- Definir regras de mapeamento do modelo lógico NoSQL baseado em agregados para o modelo de Documento, Colunar e Chave-Valor;
- Definir o processo de projeto físico NoSQL que controla a execução dessas regras de mapeamento e implementar na ferramenta BrModeloNext;
- Avaliar a implementação desenvolvida.

## 1.5 Justificativa

Apesar dos benefícios de não se especificar um esquema serem muitos, como por exemplo, a flexibilidade, diversos projetos de BD se beneficiariam da possibilidade

de especificar um esquema em BD NoSQL de antemão, pois dessa forma é possível ter um controle dos dados e modelá-los de uma forma mais adequada ao modelo de negócios, mantendo a flexibilidade e uma disposição concisa entre os dados.

Assim, é desejável disponibilizar uma ferramenta que aproveite o conhecimento dos projetistas de BD baseados em modelagens conceituais e que possibilite portar tais esquemas para BD NoSQL. Visto que não foi encontrado nenhum trabalho relacionado a esta proposta e, portanto, existe uma lacuna de pesquisa nessa área, é justificável o desenvolvimento do presente trabalho de forma a agregar a plataforma BrModeloNext um método simples de conversão que possibilita definir estruturas em BD NoSQL específicas para cada tipo de problema.

*Figura 1 - Ranking DB NoSQL*

| Rank | | | DBMS | Database Model | Score | | |
|---|---|---|---|---|---|---|---|
| Dec 2017 | Nov 2017 | Dec 2016 | | | Dec 2017 | Nov 2017 | Dec 2016 |
| 1. | 1. | 1. | Oracle | Relational DBMS | 1341.54 | -18.51 | -62.86 |
| 2. | 2. | 2. | MySQL | Relational DBMS | 1318.07 | -3.96 | -56.34 |
| 3. | 3. | 3. | Microsoft SQL Server | Relational DBMS | 1172.48 | -42.59 | -54.17 |
| 4. | 4. | 4. | PostgreSQL | Relational DBMS | 385.43 | +5.51 | +55.41 |
| 5. | 5. | 5. | MongoDB | Document store | 330.77 | +0.29 | +2.09 |
| 6. | 6. | 6. | DB2 | Relational DBMS | 189.58 | -4.48 | +5.24 |
| 7. | 7. | ↑ 8. | Microsoft Access | Relational DBMS | 125.88 | -7.43 | +1.18 |
| 8. | ↑ 9. | ↑ 9. | Redis | Key-value store | 123.24 | +2.05 | +3.34 |
| 9. | ↓ 8. | ↓ 7. | Cassandra | Wide column store | 123.21 | -1.00 | -11.07 |
| 10. | 10. | ↑ 11. | Elasticsearch | Search engine | 119.78 | +0.37 | +16.51 |

Fonte: https://db-engines.com/en/ranking

No caso das escolhas dos BDs NoSQL para este trabalho, eles foram selecionados segundo o nível de popularidade, como mostrado na Figura 1 com o ranking dos bancos de dados e a classificação de seus tipos. Assim o *MongoDB* foi escolhido como representante do tipo de documento e *Cassandra* e *Redis* como colunar e chave-valor, respectivamente.

## 1.6 Metodologia

Esta pesquisa foi do tipo exploratória, de forma que os procedimentos foram classificados como experimentais.

Assim sendo, a primeira etapa visava obter informações sobre o estado da arte a respeito de “Métodos para Conversão de Modelo Lógico NoSQL para Modelo Físico”. Para tal, foram feitas leituras de artigos, livros e outros trabalhos acadêmicos e de mercado a fim de entender o problema e o universo que envolve o mesmo.

Na segunda etapa foram definidas as regras de mapeamento. Para tal foi necessário um extenso estudo dos trabalhos prévios que envolvem modelos lógicos, junto com a criação e validação das regras.

Na terceira etapa foi desenvolvido o processo de projeto físico utilizando as regras descritas na segunda etapa de acordo com o BD NoSQL escolhido.

Na quarta etapa foi feita a análise e o projeto das tecnologias e recursos dos BDs necessários para implementação na ferramenta BrModeloNext.

Na quinta etapa foi feito um estudo de como avaliar a ferramenta e a execução desta avaliação.

## 1.7 Estrutura do Trabalho

O presente trabalho se divide em 5 capítulos, dos quais o primeiro apresenta os objetivos, o método utilizado e a justificativa para o desenvolvimento do TCC. O segundo capítulo apresenta uma síntese dos fundamentos teóricos que servem como alicerce para a compreensão do trabalho como um todo. Logo em seguida, no terceiro capitulo apresenta-se, o projeto, com a definição da arquitetura, das regras e a implementação envolvida. O quarto capítulo introduz os testes e resultados obtidos no projeto e, por fim, o quinto capítulo apresenta as conclusões.

# 2 Fundamentação Teórica

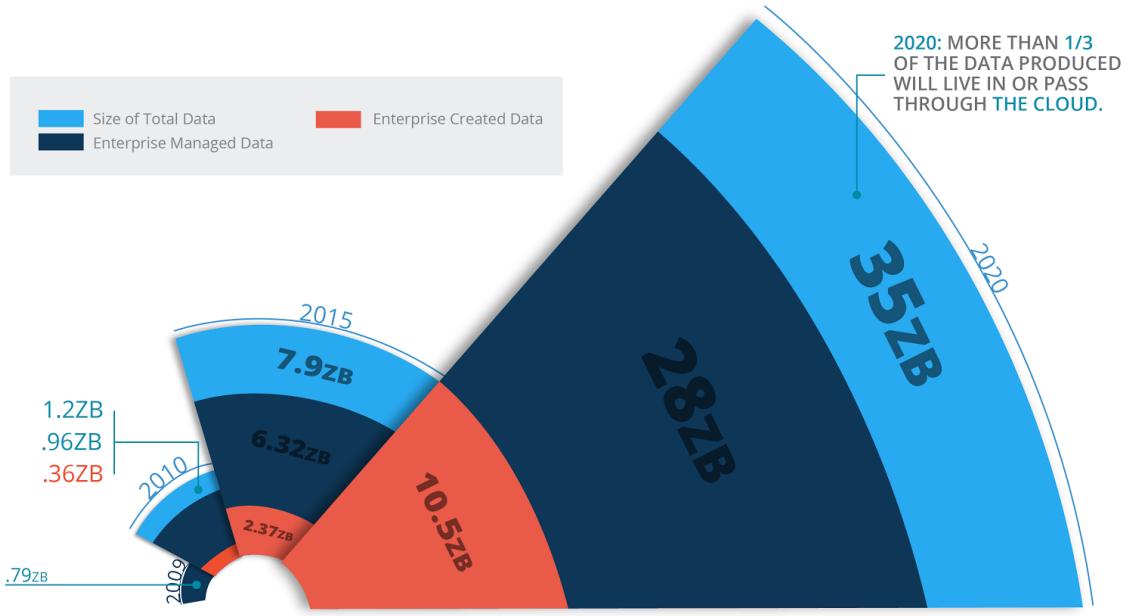
Este capítulo apresenta uma breve explicação dos conceitos essenciais para o desenvolvimento deste trabalho. Os assuntos abordados são *Big Data*, NoSQL, *Projetos de Bancos de Dados*, *projetos lógicos de bancos de dados* NoSQL e a ferramenta BrModeloNext.

## 2.1 Big Data

*Big data* é o termo comumente aplicado a grandes volumes de dados, mas que na verdade tem sua origem em tipos de dados que não conseguem ser tratados por bancos de dados tradicionais (NAVATHE&ELMASRI, 2017). Esses tipos de dados podem ter definições diferentes dependendo da indústria, de como são usados os dados e o passado dos mesmos. Assim, em 2011, o grupo Gartner definiu o que são os principais pilares da *BigData*, os 3v's (SADALAGE, 2012): *velocidade*, *variedade* e *volume*.

Primeiramente, o *volume* se refere justamente a grande quantidade de dados que são utilizados. O que era medido em gigabytes ($10^9$ bytes) se tornou zettabytes($10^{21}$ bytes) e ao passo que se encaminha para yottabytes($10^{24}$ bytes). Como mostrado na Figura 2, até 2020 a previsão é que 35 Zettabytes de dados vão ser processados em comparação com apenas 1,2 Zettabytes produzidos em 2010. Grande parte desse aumento se deve ao crescimento de dispositivos conectados com o fenômeno da IoT (DEVAN, 2017)

*Figura 2- Volume de Dados*




Fonte: (DEVAN, 2017)

*Velocidade,* segundo pesquisas da Mckinsey & Company (BAUNACH, 2012), se refere a velocidade com que um dado é criado, processado, acumulado, absorvido e processado. Ou seja, é a velocidade que um dado é gerado e movido de um ponto para o outro (DEVAN, 2017); (NAVATHE&ELMASRI, 2017).

Já a *Variedade* se define com as diferentes formas de dados disponíveis. Bancos de dados tradicionais lidam com dados estruturados, bem definidos, mas a realidade dos dados gerados hoje é que, em sua grande parte, são, dados com uma infinidade de formatos diferentes com pouca ou nenhuma estrutura (NAVATHE&ELMASRI, 2017); (DEVAN, 2017)

Com o avanço das pesquisas sobre *BigData*, foram definidos por Almeida (2017), mais 2 desafios que formam os 5v's da BigData: *Veracidade* e *Valor*.

*Figura 3 - 5 V's da Big Data*


Fonte: (ALMEIDA, 2017)

*Veracidade* se refere à confiança na qualidade dos dados, ou seja, que eles representam realmente o que está sendo passado. Para isso é necessário aplicar diversos processamentos para filtrar os dados com ruídos (DEVAN, 2017), como por exemplo números que fogem muito da média e do desvio padrão que são descartados ou campos com valores em brancos.

*Valor* se refere o resultado final obtido no processo de aplicar os 4v's. É tornar o dado valioso para quem o usa. (ALMEIDA, 2017).

A Figura 3 apresenta um resumo dos conceitos explicados.

## 2.2 NoSQL

O termo NoSQL foi aplicado primeiramente em 1998 para BD que não utilizavam SQL como linguagem de busca (STRAUCH, 2009). Posteriormente, com o avanço em pesquisas e no uso desse tipo de banco, o termo NoSQL teve sua definição atualizada para "Not Only SQL", ou seja, sua definição abraçou não só os bancos que não utilizavam integralmente a linguagem SQL, mas também bancos que não seguem o modelo de dados relacional.

Esse movimento NoSQL foi motivado por diversos fatores. Dentre eles o fato de *evitar complexidades desnecessárias* como o caso de diversas restrições impostas pelos bancos relacionais. Ao fornecerem *alta vazão (High Throughput)* com sistemas de transações mais flexíveis e formas assíncronas de sincronização de dados, os BD

NoSQL se adaptam muito melhor às necessidades de bilhões de escritas diárias em um mundo altamente conectado (BUGIOTTI, 2014).

Outros benefícios também associados a bancos NoSQL são em *escalabilidade horizontal,* que possibilita a alta disponibilidade dos bancos*, proximidade das estruturas de dados com as linguagens de programação***.**

Essas estruturas de dados mais flexíveis, retiram o trabalho do desenvolvedor de mapear estruturas para a aplicação e vice-versa. Isso leva esses bancos a possuírem características bem especificas como não requererem esquemas, o que leva em mais flexibilidade ao trabalhar com as estruturas de dados.

Em contraponto os BD NoSQL possuem uma linguagem de busca muito menos expressiva do que SQL (NAVATHE&ELMASRI, 2017), pois em grande parte o que se precisa são de operações CRUD ou SCRUD feitas por API, de forma que leva a avaliar o fator de buscas de busca dos dados antes de optar por um BD tradicional ou NoSQL.

## 2.3 Tipos de Modelos de Dados NoSQL

O modelo de dados é um dos primeiros aspectos em que um banco NoSQL se difere dos BDs tradicionais, visto que os tradicionais se baseiam-se no modelo relacional, que é composto de atributos, tuplas e relações (NAVATHE&ELMASRI, 2017). Normalmente, tratadas como uma tabela, as relações representam o domínio dos dados que estão se relacionando. Os atributos são como colunas, que representam conjuntos de dados desse mesmo grupo e as tuplas são os valores em si. Além disso, a estrutura de um BD relacional é rígida, ou seja, todas as tuplas devem possuir o mesmo conjunto de propriedades (atributos).

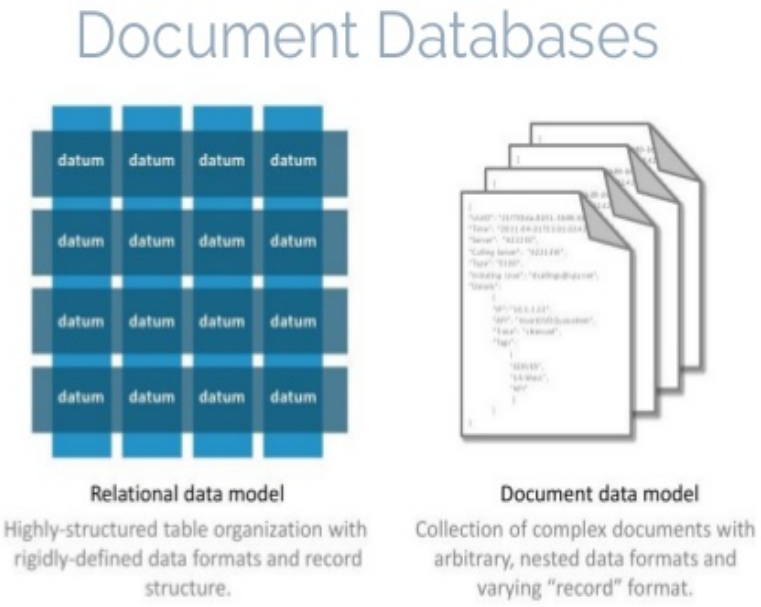
Já, os modelos de dados para NoSQL são bem menos restritivos e se dividem em 4 principais classificações definidas por LEAVITT (2010): documento, colunar, chave-valor e grafos.

O modelo de grafos, por não ser um modelo baseado em agregados não foi descrito nessa seção.

### 2.3.1 Modelo de Documento

É o modelo que trata os dados como documentos. Os documentos possuem dados em formatos como JSON (JavaScript Object Notation) ou BSON (Binary JSON), que representam uma forma intuitiva de se trabalhar com os dados ao lado da programação orientada a objeto. Um comparativo entre o modelo relacional e o modelo de dados orientado a documento pode ser observado na Figura 4 (MONIRUZZAMAN e HOSSAIN, 2013)

*Figura 4 - Relacional VS Documento*




Fonte: (MONIRUZZAMAN e HOSSAIN, 2013)

Nessa estrutura cada documento possui um identificador e atributos com valores do tipo string, array ou até outro documento agregado. Além disso, seus esquemas são dinâmicos, ou seja, cada documento pode apresentar estruturas diferentes dos demais. Um dos bancos mais populares do tipo documento é o MongoDB, que é abordado neste trabalho.

Além disso, este é o modelo que possui a melhor capacidade de busca entre todos os outros modelos, sendo que cada atributo pode ser referenciados na busca(evidenciado principalmente no MongoDB) (MONGODB, 2016).

### 2.3.2 Modelo Colunar

O modelo colunar grava e processa dados por coluna ao invés de linha, como nos bancos de dados relacionais. Neste caso, cada registro pode apresentar um conjunto de colunas diferente e mesmo apresentar colunas compostas por outras colunas. Teve sua origem a partir de análises das quais concluíram que operações com colunas apresentavam um grande poder de processamento paralelo. O precursor desse modelo foi o BigTable, do Google, que era definido como “um sistema de armazenamento para manipulação de dados estruturados feito para escala muito grande de tamanho”. (STRAUCH, 2009), ou seja, ele foi criado especificamente com o intuito de trabalhar com grandes volumes de dados.

O banco abordado nesse trabalho é o Cassandra, e este apresenta várias características oriundas de seus predecessores, o BigTable e o Dynamo, da Amazon. Seu modelo, o colunar apresenta uma estrutura mais simples se comparada ao documento e poder de busca igualmente inferior, com os índices abrangendo chaves primárias e atributos.

### 2.3.3 Modelo Chave-Valor

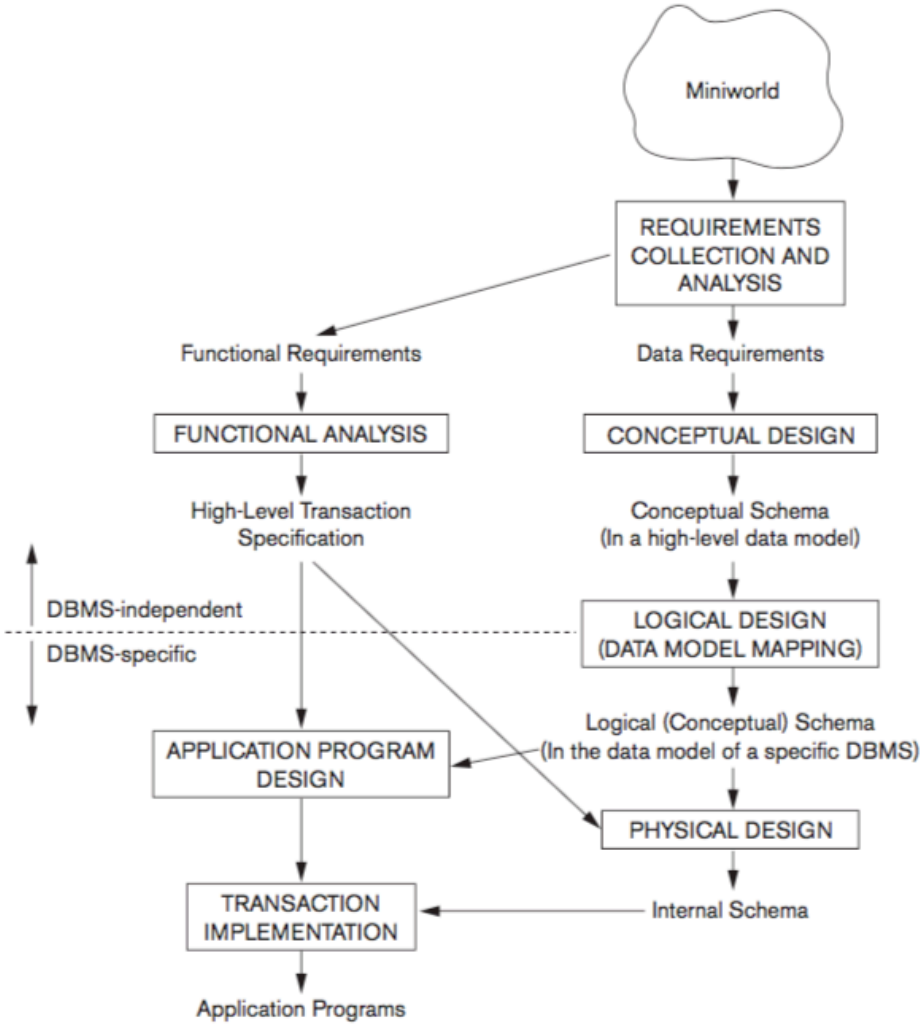
É o modelo mais simples entre todos os modelos, sendo compostos basicamente de uma chave indexada e um valor (dicionário), com as buscas se limitando a os índices das chaves primárias. Tem sua maior aplicação em modelos de negócios que necessitam de acesso instantâneo a uma informação, pois a estrutura não suporta consultas complexas ou validações em cima dos dados (STRAUCH, 2009). Além disso, suportam alta escalabilidade e ,em alguns casos, todo o banco é manipulado em memória, como é o caso do Redis, banco tratado nesse trabalho. (NELSON, 2016)

## 2.4 Projetos de Bancos de Dados

O projeto de bancos de dados é uma parte muito importante no desenvolvimento de uma aplicação como um todo. Ele engloba todas as etapas para se entender o problema da aplicação, adequar os problemas às tecnologias disponíveis e, por fim, implementa-lo e colocar em operação, a fim de finalizar o projeto de BD.

Segundo Heuser (1998), existem 3 etapas para o Projeto de bancos de dados: projeto conceitual, projeto lógico e, por fim, projeto físico. Essas etapas são ilustradas na Figura 5, a partir de uma Analise de Requisitos. Logo em seguida com os requisitos coletados, é feita a modelagem conceitual (Conceptual Design). A partir desses dados é possível gerar um projeto Lógico (Logical Design) e, no final do processo, tem-se a parte da implementação que, é o projeto físico (Physical Design). Estas etapas são descritas nas seções 2.4.1, 2.4.2 e 2.4.3 com a ótica voltada para bancos relacionais, em contraponto com a seção 2.4.4 que é voltada para os SGBDS NoSQL.

*Figura 5 - Projeto de Banco de Dados*




Fonte: (NAVATHE&ELMASRI, 2017)

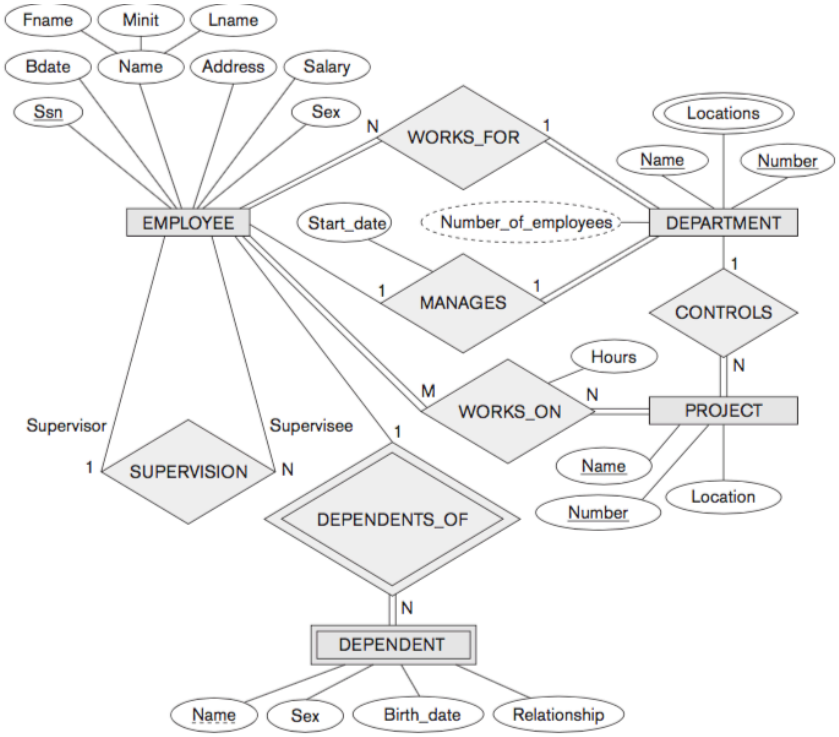
### 2.4.1 Modelagem Conceitual

A modelagem conceitual é a primeira etapa de um projeto de BD, e consiste em especificar um modelo conceitual de BD. O modelo conceitual é definido a partir de um levantamento de requisitos de dados da aplicação e produz, em geral, um diagrama entidade-relacionamento (HEUSER, 1998).

No levantamento de requisitos o projetista de BDs entrevista os futuros usuários do BD para entender e documentar os requisitos de todos dados que envolvem o banco (NAVATHE&ELMASRI, 2017). Podem ser feitas uma ou várias entrevistas com diversos usuários para chegar nos requisitos funcionais da aplicação.

Uma vez encontrados esses requisitos é feito o diagrama ER (entidade-relacionamento), que pode ser visto na Figura 6. Nele é feito o detalhamento usando a notações para a descrição das entidades, do mundo real e suas relações e propriedades. Essa é uma descrição de nível mais elevado, sendo que não são levados em conta detalhes físicos da implementação do BD.

*Figura 6- Diagrama ER*

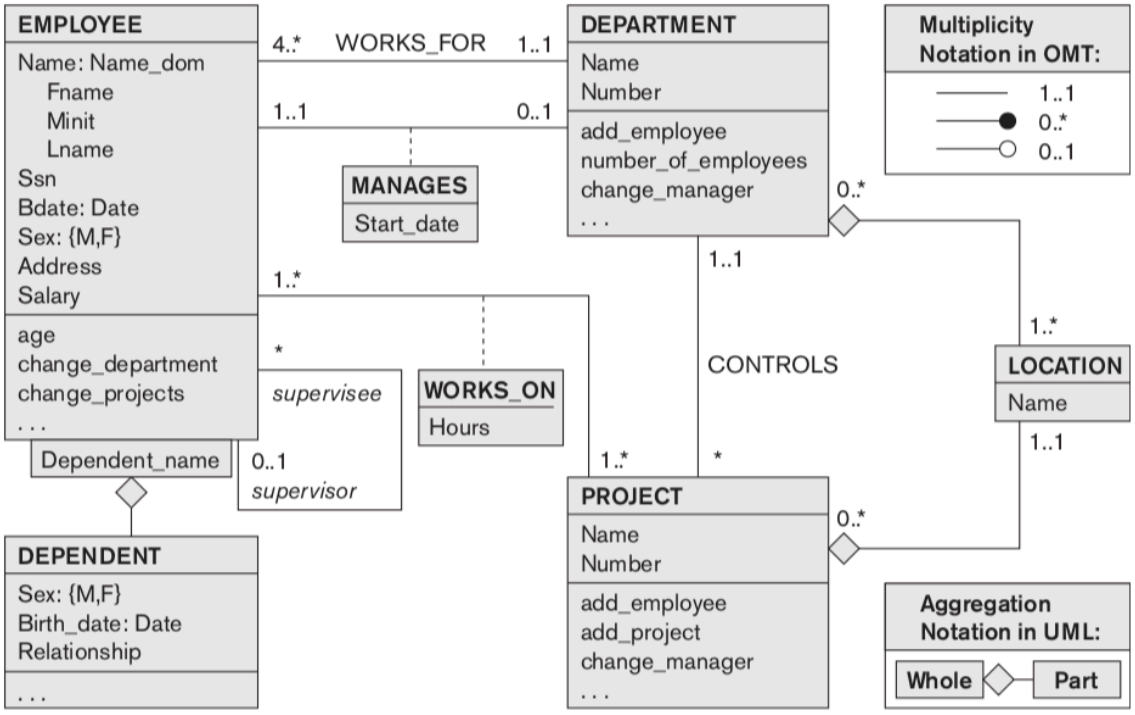



Fonte: (NAVATHE&ELMASRI, 2017)

## 2.4.2 Projetos Lógico

O projeto lógico é o divisor entre a dependência ou não de um sistema de BD. Nele são descritos os dados com o máximo de detalhes possíveis, sem se preocupar com aspectos físicos. São aplicadas restrições aos objetos, atributos e relacionamentos, visto que antes o que era feito por um modelo genérico na etapa conceitual, passa a portar limitações de integridade, normalização, entre outros. Assim , ele representa a estrutura de dados de um BD, conforme visto por usuários do SGBD. (HEUSER, 1998) .

*Figura 7 – Exemplo Projeto Lógico*




Fonte: (NAVATHE&ELMASRI, 2017)

Como exemplo, a Figura 7 mostra o mesmo domínio apresentado na Figura 6, porém considerando agora aspectos lógicos como a relação e tipo dos dados entre outros. Essa etapa também é muito importante para observar e fazer melhorias no modelo, de forma que é possível mudar generalizações e regras de limitações nos atributos.

## 2.4.3 Projetos Físico

Nesse nível, as representações dos dados estão intimamente ligadas ao Sistema de Gerênciamento de Banco de Dados (SGBD) e com as especificações impostas pelo mesmo, sendo que cada SGBD pode definir um modo diferente de implementação física das características e recursos necessários para armazenamento e manipulação das estruturas de dados (COUGO, 1997).

O projeto físico representa como a informação vai ser construída no BD, com a especificação de todos os detalhes sendo feita pela interface suportada por cada banco. Por exemplo, em bancos tradicionais, como MySQL, as especificações são

feitas em SQL, diferentemente de bancos NoSQL, como MongoDB, que se pode especificar o projeto a partir de sua API em JavaScript.

*Figura 8 - Exemplo Projeto Físico*

```
CREATE TABLE EMPLOYEE
        ( Fname                 VARCHAR(15)             NOT NULL,
          Minit                 CHAR,
          Lname                 VARCHAR(15)             NOT NULL,
          Ssn                   CHAR(9)                 NOT NULL,
          Bdate                 DATE,
          Address               VARCHAR(30),
          Sex                   CHAR,
          Salary                DECIMAL(10,2),
          Super_ssn             CHAR(9),
          Dno                   INT                     NOT NULL,
        PRIMARY KEY (Ssn),
CREATE TABLE DEPARTMENT
        ( Dname                 VARCHAR(15)             NOT NULL,
          Dnumber               INT                     NOT NULL,
          Mgr_ssn               CHAR(9)                 NOT NULL,
          Mgr_start_date        DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber               INT                     NOT NULL,
          Dlocation             VARCHAR(15)             NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE PROJECT
        ( Pname                 VARCHAR(15)             NOT NULL,
          Pnumber               INT                     NOT NULL,
          Plocation             VARCHAR(15),
          Dnum                  INT                     NOT NULL,
        PRIMARY KEY (Pnumber),
        UNIQUE (Pname),
        FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE WORKS_ON
        ( Essn                  CHAR(9)                 NOT NULL,
          Pno                   INT                     NOT NULL,
          Hours                 DECIMAL(3,1)            NOT NULL,
        PRIMARY KEY (Essn, Pno),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );
CREATE TABLE DEPENDENT
        ( Essn                  CHAR(9)                 NOT NULL,
          Dependent_name        VARCHAR(15)             NOT NULL,
          Sex                   CHAR,
          Bdate                 DATE,
          Relationship          VARCHAR(8),
        PRIMARY KEY (Essn, Dependent_name),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
```

Fonte: (NAVATHE&ELMASRI, 2017)

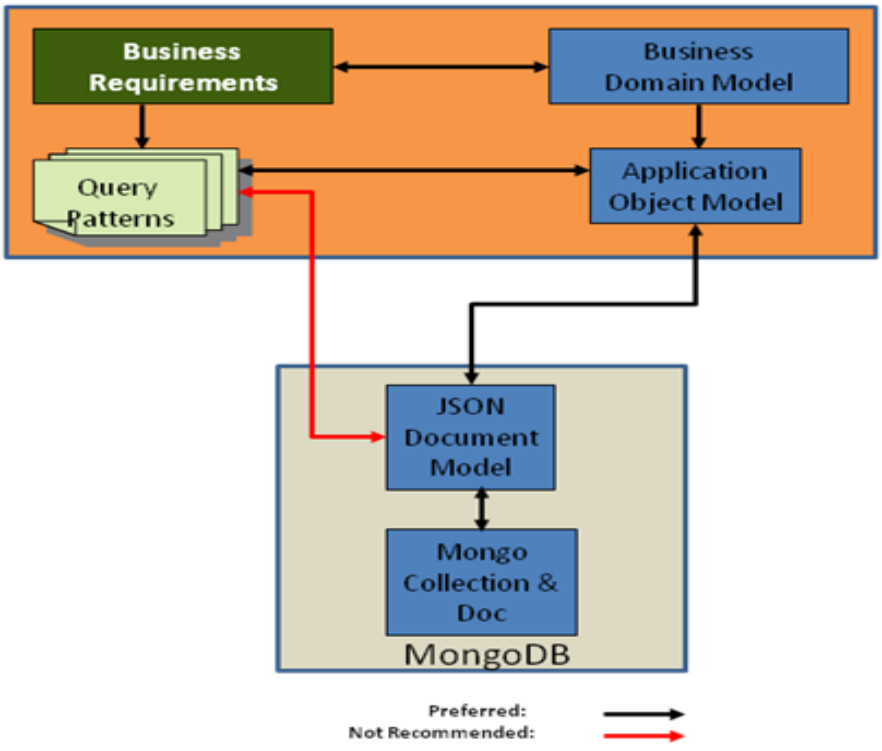
É possível exemplificar essa representação através do código SQL exibido na Figura 8, que é a representação física do projeto lógico mostrado na Figura 7 da seção anterior.

### 2.4.4 Projetos Lógicos de Bancos de dados NoSQL

As técnicas de projeto voltadas para bancos de dados relacionais estão bem consolidadas (HSIEH, 2014). Técnicas, como mapeamento lógico para físico e normalização, são aplicadas com segurança por profissionais em todo mundo. Em contraponto, os bancos de dados NoSQL não gozam de mesmo privilégio, visto que grande parte da modelagem é aplicada diretamente na modelagem física, sem passar pelas outras etapas, como os bancos de dados tradicionais.

A Figura 9 exibe uma das formas em que é feita a modelagem de bancos de dados NoSQL do tipo documento. O primeiro passo é obter os requisitos do negócio e o modelo do domínio. A seguir se definem que tipos de consultas vão ser feitas, a fim de saber como organizar esses dados e buscá-los mais facilmente. Paralelamente, é descrito o modelo de dados que a aplicação irá utilizar. Por fim, com essas informações é possível definir o modelo do documento JSON que vai ser utilizado no BD.

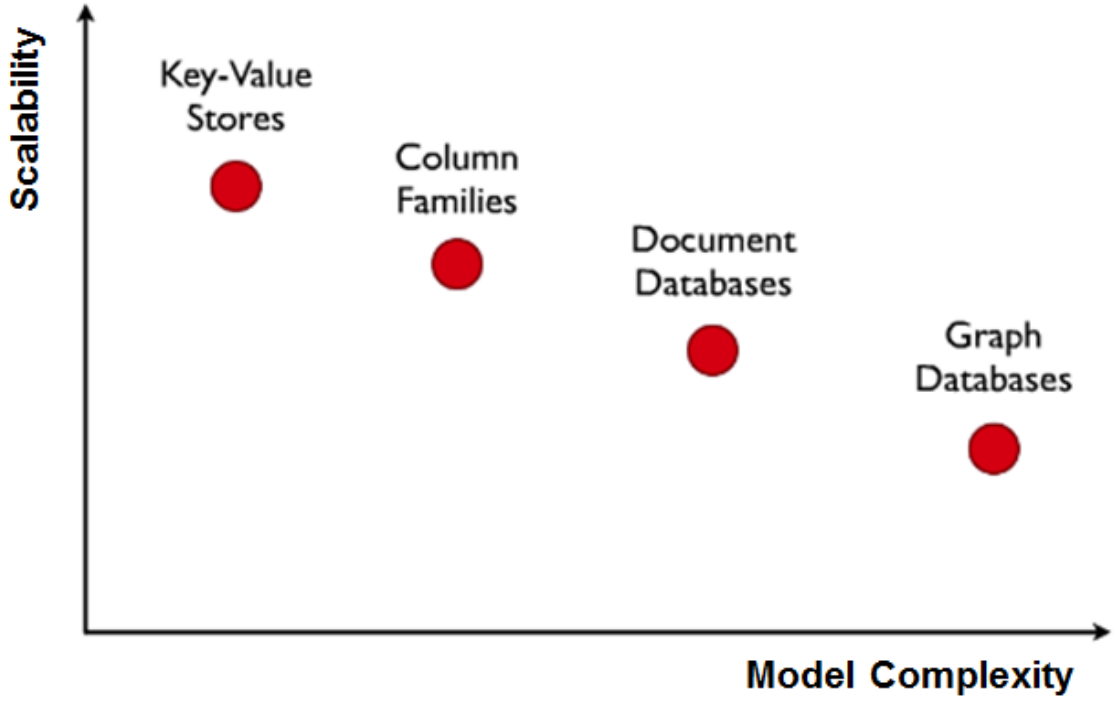
*Figura 9 - Modelagem Padrão para Bancos de Documento*




*Fonte: (HSIEH, 2014)*

Muitos autores atestam que o desenvolvimento de métodos para modelagem de alto nível e suporte para bancos NoSQL são necessários. Assim surgiram pesquisas como o NoAm (NoSQL Abstract Model) de BUGIOTTI (2014), que visa explorar o modelo de dados como um modelo abstrato em nível lógico, do qual este pode ser aplicado a alguns bancos de dados. Outro modelo, do qual é baseado este trabalho, é o de LIMA (2016). Ele enfatiza em projeto lógico de bancos de dados para documentos, visto que apesar deste modelo focar especificamente em bancos do tipo documento, é possível dada a abrangência do modelo de documentos converter esse modelo para tipos menos complexos, como colunar e chave valor, do qual é mostrado na Figura 10, que exibe o nível de complexidade de cada modelo.

*Figura 10 - Complexidade modelos NoSQL*




Fonte: (HSIEH, 2014)

O projeto lógico de bancos de dados NoSQL na literatura se baseia fortemente no conceito de agregados, que é explicado na próxima seção.

### 2.4.5 Agregados

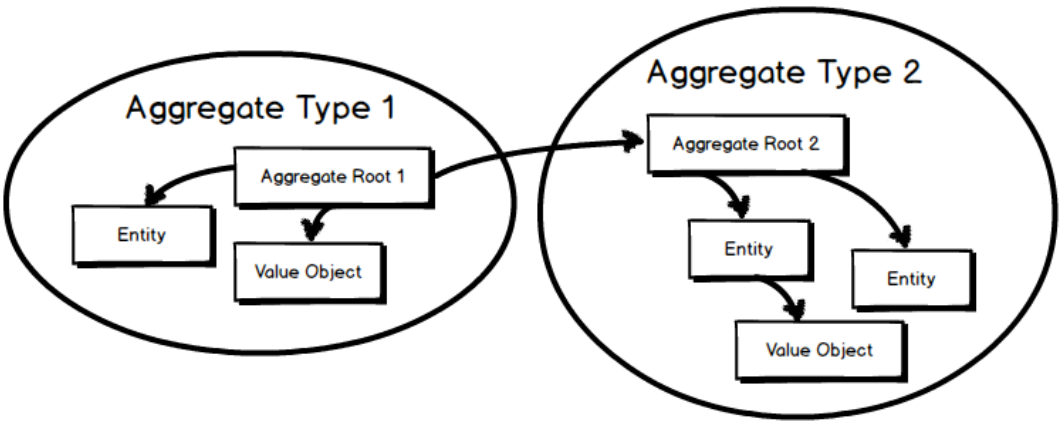
Tanto BUGIOTTI (2014), quanto LIMA (2016), se baseiam fortemente no conceito de agregados. Agregado é um paradigma que se encaixa muito bem para BDs NoSQL pelo fato de agrupar dados que se relacionam, de forma que são acessados tudo de uma só vez (FOWLER, 2012). Além disso, é possível que seja visualmente mais fácil para o desenvolvedor, que possui toda a informação necessária em um único documento, diferente de BDs relacionais em que as informações estão dispersas em varias tabelas e precisam ser unidas a fim de se ter o mesmo resultado.

Basicamente, a modelagem de agregados trata o domínio do problema como entidades. Entidades são classificadas como raiz ou local.

As entidades raízes são as que estão visíveis para as outras entidades, ou seja, a entidade raiz possui um identificador único que pode ser trocado com outras entidades.

Já as entidades locais, estão contidas nas entidades raízes e não podem ser referenciadas fora dos limites das entidades raiz.

*Figura 11 - Relacionamento Agregados*




*Fonte : VERNON (2014)*

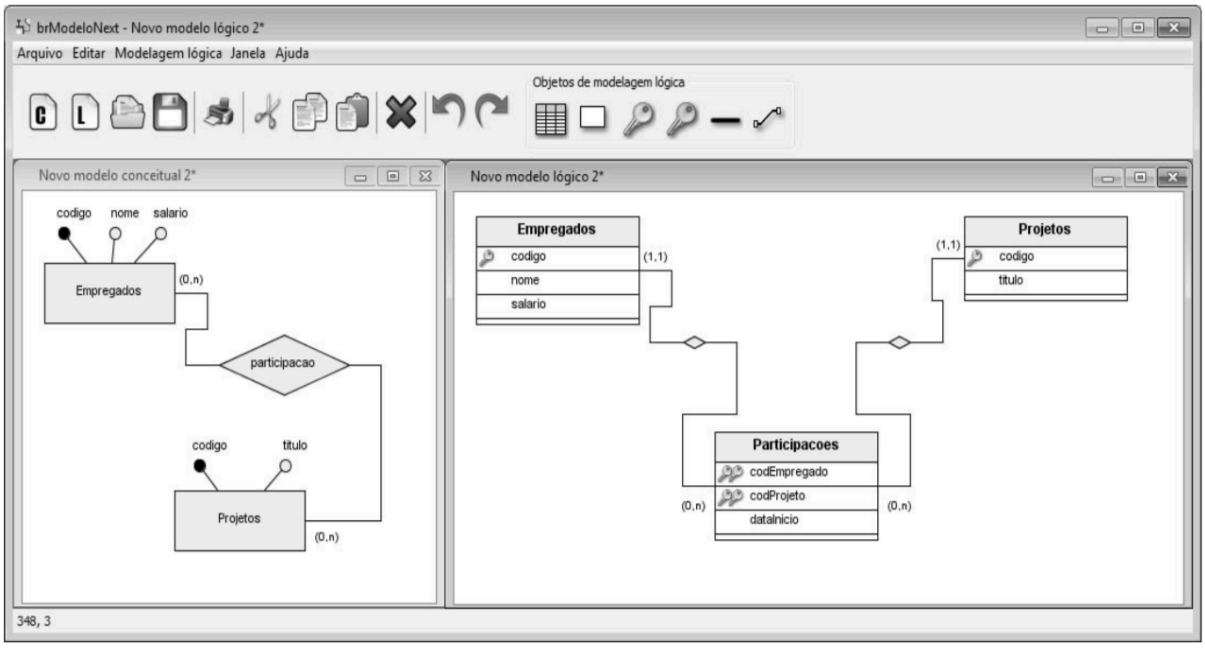
Na Figura 11, como pode ser visto, apenas o agregado raiz (Aggregate Root 1) do grupo de agregados tipo (Aggregate Type 1) possui uma referência do agregados raiz (Aggregate Root 2) 2 do grupo de agregados tipo (Aggregate Type 2), de forma que as entidades locais não podem ser referenciadas.

## 2.5 Ferramenta BrModeloNext

A ferramenta BrModeloNext (MENNA, RAMOS e MELLO, 2011) é uma extensão da ferramenta BrModelo, que foi criada pelo Grupo de BD da UFSC com o intuito de ter uma ferramenta que suportasse o ensino de modelagem relacional de BDs.

A ferramenta BrModelo suporta a geração do modelo conceitual, lógico e a geração do código SQL referente ao modelo físico, com a notação Entidade e Relacionamento Estendido (HEUSER, 1998). Entretanto, devido a limitações relacionadas a sua usabilidade e uso restrito à plataforma Windows, foi desenvolvida uma a nova versão da ferramenta, a BrModeloNext, que resolve essas limitações da versão anterior e inclui novas funcionalidades, como a possibilidade de visualizar várias modelagens ao mesmo tempo. Um exemplo da interface gráfica da BrModeloNext é mostrado na Figura 12 (MENNA, RAMOS e MELLO, 2011).

*Figura 12 - Modelagem Lógica brModeloNext*




Fonte: (MENNA, RAMOS e MELLO, 2011)

### 2.5.1 Notação Modelo Lógico NoSQL no BrModeloNext

Recentemente, a BrModeloNext foi estendida (MENNA, RAMOS e MELLO, 2011) para permitir a geração de uma modelagem lógica baseada em agregados, a

partir de uma modelagem conceitual ER, visando o projeto lógico de bancos de dados NoSQL. O modelo lógico proposto é definido com o seguinte conjunto de componentes: Coleções, blocos e atributos, onde os blocos são documentos ou objetos aninhados dentro das coleções, os quais podem ter disjunção com outros blocos. Os atributos pertencem a blocos ou coleções e possuem papel similiar aos atributos de uma tabela do modelo ER, sendo que estes podem ser identificadores ou de referência. Exemplos desse modelo podem ser vistos nas Figuras 19 e 34.

Para que a ferramenta BrModeloNext possua a funcionalidade completa de uma ferramenta de projeto para bancos de dados NoSQL, é desejável que a partir do modelo lógico sejam definidos os modelos físicos para os tipos de bancos NoSQL orientados a agregados, sendo esta a proposta desse trabalho.

## 2.6 JSON

JSON (Javascript Object Notation) é uma estrutura de dados derivada de literais da linguagem JavaScript, porem com especificações que o tornam independente de linguagem, podendo ser usado com facilidade em outras linguagens (ECMA-404, 2014). Tem como objetivos ser: (MOZZILLA, 2016)

- Mínimo
- Portável
- Textual
- Subconjunto do JavaScript

Sua estrutura é baseada em documentos de chave-valor que podem ter outros documentos aninhados (objetos) ou outros tipos como Strings, Numbers, Boolean ou Arrays.

*Figura 13 - Exemplo Documento JSON*

```
{
  "Image": {
      "Width":  800,
      "Height": 600,
      "Title":  "View from 15th Floor",
      "Thumbnail": {
          "Url":    "http://www.example.com/image/481989943",
          "Height": 125,
          "Width":  100
      },
      "Animated" : false,
      "IDs": [116, 943, 234, 38793]
    }
}
```

Fonte: (ECMA-404, 2014)

A Figura 13 mostra uma estrutura de dados definida em JSON. Nela a chave *Image* representa um documento que contem as chaves *Width* e *Height* do tipo Number, Title do tipo String, Thumbnail como um objeto aninhado, *Animated* como Boolean, e por fim, ID's como Array.

## 2.7 JSON Schema

Um esquema é uma definição da estrutura de um conjunto de dados que estão contidos no BD. Ele dá flexibilidade ao projetista de abstrair os dados do banco de forma que se possa observar o comportamento dos dados no sistema.

Usado em grande parte pelos bancos de dados tradicionais, o esquema tem grande papel de facilitar a visualização das relações entre as entidades e definir aspectos de restrições e tipos em níveis mais baixos de abstração (NAVATHE&ELMASRI, 2017).

Com o crescimento do formato JSON pelo uso em diversas aplicações na Web, surgiu a necessidade de especificar o formato esperado de dado em JSON, criando, por conseguinte, o JSON Schema.

*Figura 14 - Exemplo JSONSchema*

```
{
    "title": "Person",
    "type": "object",
    "properties": {
        "firstName": {
            "type": "string"
        },
        "lastName": {
            "type": "string"
        },
        "age": {
            "description": "Age in years",
            "type": "integer",
            "minimum": 0
        }
    },
    "required": ["firstName", "lastName"]
}
```

*Fonte: (ORGANIZATION, 2017)*

O JSON Schema possui diversas palavras-chaves para garantir a restrição no documento a ser validado. Na Figura 14 é possível ver um esquema, para uma coleção de documentos JSON. A palavra-chave *properties* define as propriedades da coleção, no caso os atributos dos documentos JSON.

É possível, a partir de cada propriedade, restringir um tipo para aquele atributo com a palavra-chave *type*, adicionar cardinalidades como *maxItems* que define um número máximo de objetos ou adicionar valores mínimos para inteiro como é o caso de *age* com a palavra-chave *minimum, mostrada na Figura 14*.

*Figura 15 - Exemplo Instancia*

```
{
  "firstname": "Nathan",
  "lastname": "Reuter Godinho",
  "age": 25
}
```

*Fonte: Gerada pelo Autor*

Propriedades podem se tornar obrigatória usando a palavra-chave *required,* de forma que um documento JSON só é válido se dentro dele estiver contido cada

propriedade listada no atributo *required*. A Figura 15 mostra uma instância JSON preenchida conforme o esquema da Figura 14. No caso, o atributo age pode ser omitido sem ferir as restrições impostas pelo esquema.

Outras propriedades importantes do JSONSchema são *aditionalProperties, #ref, allOf, anyOf, oneOf* e *Not.* O primeiro tem o papel de restringir ou não o acréscimo de novas propriedades além daquelas listada no esquema. Em seguida, o ref serve para referenciar outros esquemas, de forma a tornar a declaração dos esquemas mais modular e reutilizável.

*Figura 16 - Palavra-chave OneOf*

```
{
  "oneOf": [
    { "type": "number", "multipleOf": 5 },
    { "type": "number", "multipleOf": 3 }
  ]
}
```

*Fonte: Gerada pelo Autor*

Por fim o resto das propriedades tem o papel de combinar esquemas, como mostrado na Figura 16, onde só será aceito na propriedade números múltiplos de 5 ou de 3.

O JSON Schema tem seu uso explanado com mais detalhes na seção 3.1.1 como forma de representação de um esquema para mongoDB.

## 2.8 Mongo

O MongoDB é o representante da família dos bancos NoSQL do tipo documento, visto que este foi desenvolvido com foco em aplicações web e visando suportar grande quantidade de dados com grande vazão de leituras e escritas no banco além de várias estratégias a fim de facilitar a replicação.

Seu formato de dado utilizado é o BSON que é um conjunto estendido do JSON (seção 2.7.1), que acrescenta novos tipos de dados como dados binários. Tem característica

de ser leve, fácil de se efetuar buscas e eficiente quando for decodificado e codificado (SPECIFICATION, 2018).

Como interface entre usuário e banco, o MongoDB utiliza um shell interpretador de JavaScript com funções específicas implementadas para que o usuário possa fazer toda a administração trivial do banco (BANKER, 2012).

## 2.9 Cassandra

O Cassandra é o BD NoSQL escolhido para representar a família dos bancos colunares. Foi projetado para ser distribuído, descentralizado, escalável, tolerante a falha e que provê alta disponibilidade. Criado no Facebook, ele uniu o design do Dynamo da Amazon com o modelo de dados do BigTable da Google e hoje é usado por diversos sites como Twitter, Netflix (HEWITT, 2011).

## 2.10 Redis

O Redis é o BD escolhido para representar a família dos bancos de dados chave-valor. Muito popular, o Redis apresenta alto desempenho por fazer todas as operações em memória e persistir os dados quando necessário (CARLSON, 2013). Não possui grande capacidades de buscas, sendo todos os acessos feitos a partir dos índices, cabendo ao usuário administrar corretamente a fim de abstrair melhor performance para sua aplicação (RUSSO, 2010). O banco apresenta um conjunto de dados diversos, porém simples, que são os tipos strings, bitfields, streams, índices geoespaciais, hyperlog, bitmaps, list, sets, sorted sets e hashes.

# 3 Mapeamento da Modelagem Lógica de Agregados para NoSQL

Com base nos conhecimentos adquiridos e exaltados na seção 2 e além de aprendizados práticos em cima da plataforma BrModelonext, foram desenvolvidas regras de mapeamento de modelagem lógica para agregados NoSQL. Essas regras acrescentam novas funcionalidades à plataforma e a tornam a primeira plataforma a converter modelagens conceituais (alto nível) para física NoSQL.

Ao longo dessa seção são apresentadas as informações relacionadas ao projeto (seção 3.1), as regras de conversão para Documento, focadas no BD MongoDB (seção 3.2), regras de conversão para colunar, focadas no BD Cassandra (seção 3.3) e regras para BD chave-valor, focadas no Redis (seção 3.4). Nas seções seguintes são apresentadas a arquitetura (3.4) e por fim a implementação dessas funcionalidades no BrModelonext.

## 3.1 O Projeto

O mapeamento de modelagem lógica de agregados para modelo físico NoSQL é um conjunto de funcionalidades adicionais para a ferramenta BrModelonext a fim de torna-la uma ferramenta com suporte completo desde a modelagem até a geração de esquemas físicos para os modelos de bancos de dados escolhidos pelo usuário da aplicação.

Para o projeto foram escolhidos 3 modelos mais representativos no âmbito de BDs NoSQL. Os modelos Documento, Colunar e Chave-Valor, dos quais para cada um destes, foi eleito um BD, com base em popularidade e recursos adicionais, para que fossem modeladas as regras específicas para esses bancos. Essas regras são específicas para esses bancos pois são esquemas gerados em nível físico, o qual é dependente da aplicação que está recebendo, diferente de esquemas em nível conceitual e lógico que podem ser generalizados para cada caso.

### 3.1.1 Compreensão do MongoDB

No início dos estudos para o presente projeto, foi feito um levantamento de todos os recursos implementados e disponíveis no MongoDB para que se pudesse atrelar esses recursos com as funcionalidades esperadas das regras. Um dos recursos esperados por exemplo é a possibilidade de definir todo modelo de agregados apenas com as funcionalidades do banco, o qual se confirmou após as pesquisas.

O shell do MongoDB deve suportar também condições que garantam inserções corretas de coleções, dos atributos que essas coleções têm, se o tipo desses atributos estava no formato correto e entre outros casos. Além disso, é preferível que se possa avisar o usuário desses erros de validação de forma que se tenha um retorno, seja para o usuário em um terminal ou para uma aplicação.

Após estudos sobre a documentação, foram encontradas diretivas como demonstrado na Figura 17, de forma que esse o especifica uma coleção contacts, que vai receber um validador. Esse validador espera encontrar um conjunto de atributos do tipo phone, email e status podendo especificar o tipo, como no caso do phone, ou podendo definir conjuntos esperados, como é o caso de status que pode ter somente Unknown ou Incomplete para que esteja correto.

*Figura 17 - Primeira forma de Validação Mongodb*

```
db.createCollection( "contacts",
   { validator: { $or:
      [
         { phone: { $type: "string" } },
         { email: { $regex: /@mongodb\.com$/ } },
         { status: { $in: [ "Unknown", "Incomplete" ] } }
      ]
   }
} )
```

Fonte: (MONGO, 2017)

Essa forma de validação se encaixou bem para o trabalho e foi a que se baseou a primeira versão do TCC, com os casos testes de funcionalidade se comportando como o esperado.

Porém, no intervalo do desenvolvimento deste projeto, foi lançada a versão 3.6 do MongoDB com suporte a JSON SCHEMA como forma de validação primária, o que se tornou muito mais atraente de ser implementada pois é uma notação muito mais popular do que diretivas específicas para um único banco, além de ser mais legível para humanos e de implementação mais intuitiva.

*Figura 18 - Forma atual de validação MongoDB*

```
db.createCollection("book", {
  validator:{
    $jsonSchema: {
      bsonType: "object",
      properties:{
        name: { bsonType: "string"},
        _id: { bsonType: "objectId"},
        author_REF: { bsonType: "objectId"},
      },
      required: ["name", "_id", "author_REF"],
    }
  },
  validationAction: "error",
  validationLevel: "moderate"
});
```

*Fonte: Gerada pelo Autor*

Assim, a sintaxe se torna como a mostrada na Figura 18, recebendo o **$**jsonSchema como validador com todas as funcionalidades suportadas pelo mesmo especificadas na seção 2.72.

A Figura 18 também trás consigo outras duas diretivas, validation (BANKER, 2012) e validationLevel. Essas são as diretivas utilizadas para ajustar as ações que o banco vai tomar quando os dados inseridos ou modificados não seguirem as regras propostas no esquema.

Por exemplo, validationLevel pode ser moderate, strict ou nulo. No primeiro e no segundo caso, o banco procura validar todos os esquemas que vão ser inseridos no banco, com a exceção que o moderate não valida os dados inseridos antes de ser especificado o validador. Em caso de que ao inserir o dado esteja desconforme com o esquema, entra o validationAction que pode apenas avisar o usuário que está errado

ou impedir a inserção e acusar erro, que são as palavras *warning* ou *error* respectivamente.

A opção de escolher a forma de validação foi implementada e descrita com mais detalhes na seção 3.5

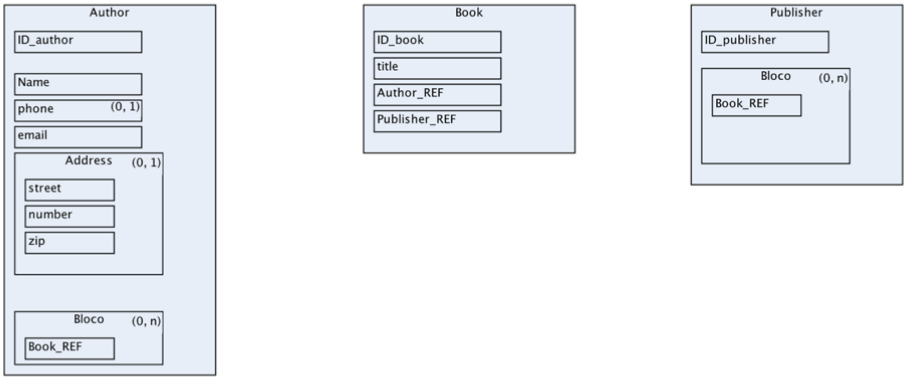
### 3.1.2 Compreensão do Cassandra

No início dos estudos, de forma análoga como foi feito no estudo da seção 3.1.1, foram pesquisados e analisados todos os recursos presentes no banco para que se pudesse criar um código que validasse todas as interações com o banco de acordo com um esquema pré-definido.

A forma de interação com o usuário no Cassandra acontece pelo shell, que utiliza um linguagem própria chamada CQL (Cassandra Query Language), uma linguagem com sintaxe similar ao SQL (presente nos bancos relacionais) porem, com varias limitações, como não suportar JOINS, GROUPBY e o uso de palavras chaves especificas da linguagem (MENG, 2014).

Após os estudos, o próximo passo foi entender como o mapeamento resultou nas regras da seção 3.3. Esse processo englobou, além de observar o modelo de agregados pelas perspectivas do colunar, fazer testes representando modelos como o da Figura 19, dentro da linguagem CQL do Cassandra, e observando como ocorria essa representação do agregados para colunar. No mesmo momento em que os testes ocorriam todas as palavras-chaves essenciais eram anotadas e suas especificidades entendidas para posterior uso nos algoritmos de conversão.

*Figura 19 - Modelo Agregados para Simulação*




Fonte: Gerada pelo Autor

*Figura 20 - Simulação Convertida para Cassandra*

```
DROP KEYSPACE simpleModel;
CREATE KEYSPACE simpleModel WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '3'};

use simpleModel;

CREATE TYPE address (
  street text,
  number text,
  zip_code int
);

CREATE TABLE publisher (
    id uuid,
    name text,
    founded timestamp,

    PRIMARY KEY (id)
);

CREATE TABLE books (
    id uuid,
    name text,
    title text,
    author uuid,
    publisher uuid,

    PRIMARY KEY (id)
);

CREATE TABLE author (
    id uuid,
    name text,
    email text,
    addresses map<text, frozen <address>>,
    books list<uuid>,

    PRIMARY KEY (id)
);
```

Fonte: Gerada pelo Autor

Na Figura 19 e na Figura 20 podem ser vistos o modelo feito no BrModeloNext e a simulação da conversão com o código em CQL, respectivamente.

### 3.1.3 Compreensão do Redis

O Redis é um BD que possui um conjunto grande e diferenciado de dados para um banco do tipo chave-valor. Porem, ele continua sendo um banco chave-valor o qual tem o foco em tornar as tarefas simples rápidas, fato o qual o mesmo é utilizado em diversas aplicações como um banco secundário para cache. Consequentemente, isso leva a dificuldades de trazer um modelo mais complexo como o de agregados, do qual é modelado inicialmente no BrModeloNext, para um modelo extremamente simplificado como o de chave-valor.

*Figura 21 - Objetos como HASHES no Redis*

```
@cli
HMSET user:1000 username antirez password P1pp0 age 34
HGETALL user:1000
HSET user:1000 password 12345
HGETALL user:1000
```

Fonte: (REDIS, 2017)

Sabendo das limitações comentadas, a primeira ação foi estudar formas de se definir um esquema com os recursos do Redis. Em estudos foram analisados os tipos para que se pudesse fazer uma aproximação do modelo de agregados. O tipo HASH, por exemplo, aparenta ser um bom candidato, por poder definir uma composição de chaves e valores, como mostrado na Figura 21, visto que é definido um objeto usuário por meio de hashes. Este usuário user:1000, apresenta nome como *username*, senha como *password* e idade como *age* (REDIS, 2017).

Porem apesar de poder abstrair e representar o modelo citado como um objeto, em sua essência, continua sendo um conjunto de chaves-valores que não possibilitam a definição de esquemas, de forma que se defina se o modelo segue o padrão pré-definido do esquema e que os valores inseridos estão de acordo.

*Figura 22 - Lua Scripting com Redis*

```
if redis.call("EXISTS", KEYS[1]) == 1 then
  local payload = redis.call("GET", KEYS[1])
  return cmsgpack.unpack(payload)[ARGV[1]]
else
  return nil
end
```

Fonte: (O'ROURKE, 2017)

A forma mais próxima de se definir esquemas é modificando a forma como o Redis lida com as suas inserções, alterando o código nativo. Toda a implementação dos comandos do banco é feita na linguagem Lua, a qual é uma linguagem interpretada que o Redis da suporte de execução em seu ambiente, podendo definir funções adicionais ou redefinir existentes.

A Figura 22, por exemplo, representa um simples script em Lua que recebe uma chave como argumento *KEY[1]* e faz uma chamada para a biblioteca do Redis, para que o mesmo execute o comando *Exists* internamente e verifique se tal chave existe no banco. Caso exista, essa chave, é executada uma segunda chamada para recuperar o valor dessa chave do qual o valor é filtrado e retornado no final da função.

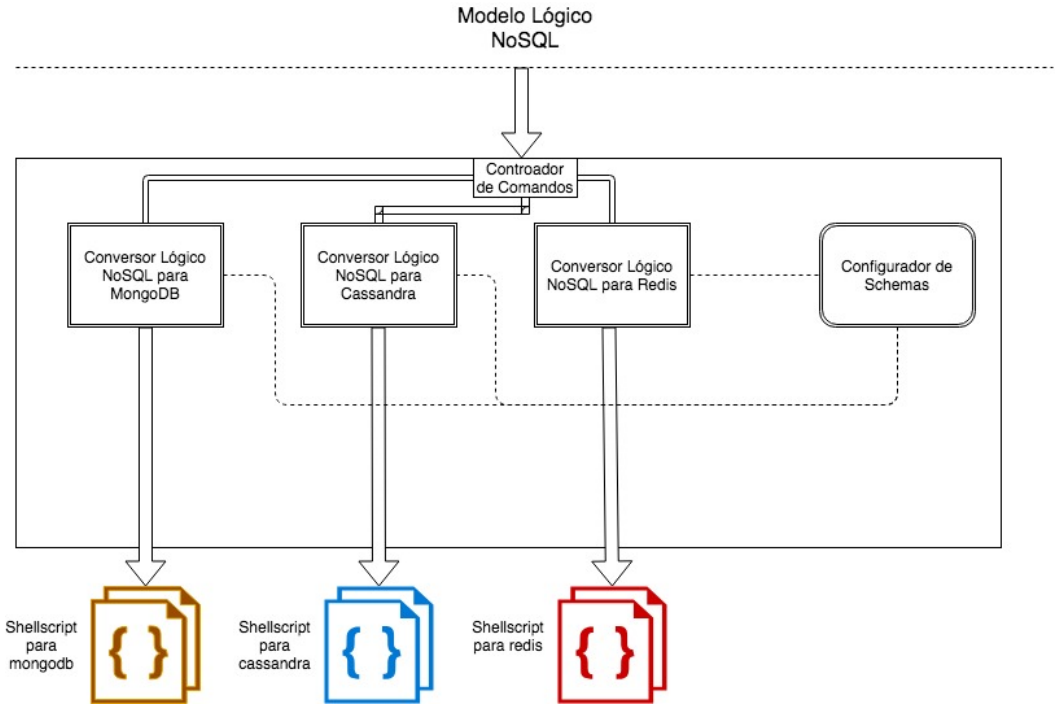
O script gerado em Lua para Redis deste presente trabalho serve como um validador simplificado para o modelo que se quer inserir, o qual retorna se o modelo está correto ou não para o esquema criado.

### 3.1.4 Arquitetura

A arquitetura representa uma visão de alto no nível do funcionamento da aplicação como um todo, com os seus elementos e suas relações que, juntos, compõe o fluxo de funcionamento do sistema. Seu objetivo é construir uma ponte entre os requisitos do negócio e os requisitos técnicos a fim de implementar os casos de usos do software.

Para o caso deste trabalho, a arquitetura do BrModeloNext com as implementações propostas funcioná da seguinte forma.

*Figura 23 - Arquitetura da Ferramenta*




Fonte: Gerada pelo Autor

A entrada da arquitetura é um modelo lógico genérico, criado por um usuário do BrModeloNext, que passou por processos de conversão de agregados(LIMA, 2016). A partir do modelo, três ações principais e uma secundária podem ser tomadas, que são selecionadas no controlador de comandos, como pode ser observado na Figura 23.

A primeira é gerar um modelo de documentos para MongoDB, que vai acionar o modelo de conversão do esquema lógico para MongoDB, convertendo o modelo para JSONS CHEMA e em seguida o adptando para instruções do Mongodb.

A segunda ação leva para geração do modelo colunar para Cassandra, que vai receber o modelo, e passar pelo modulo de Conversor Lógico NoSQL para Cassandra, que no final resultará no esquema para Cassandra.

A terceira ação leva para geração do modelo para o Redis, que passa pelo Conversor Lógico NoSQL para Redis, e retorna as instruções para gerar esquema para este banco.

Por fim, a ultima opção é o configurador de Schemas, o qual recebe diversas configurações gerais como nome do BD a ser gerado e até especifica como número de replicações.

Simplificando, a arquitetura recebe um modelo NoSQL feito para o BrModelonext e retorna instruções para Mongodb, Cassandra ou Redis.

*Tabela 1- Comparativo de Funcionalidades Atendidas Entre os Diferentes Tipos de BD's NoSQL*

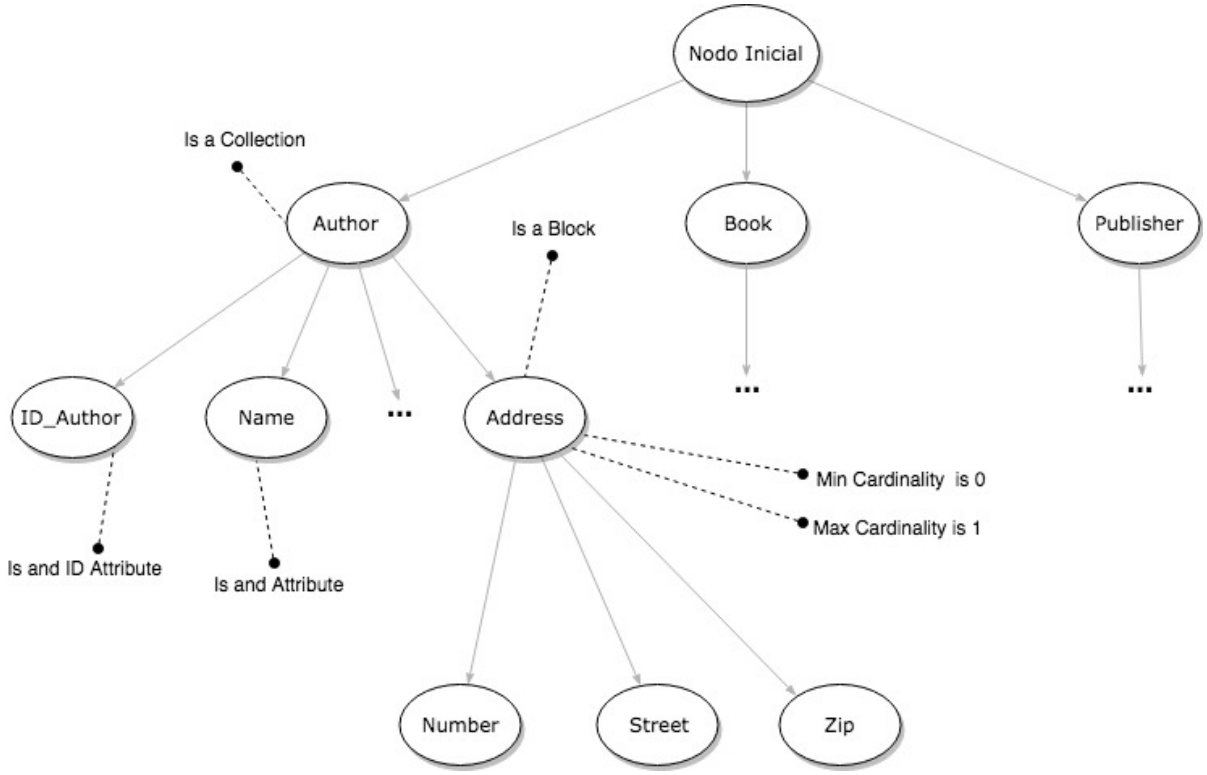
| Funcionalidades | Mongo | Cassandra | Redis |
|---|---|---|---|
| **Conversão de Coleções** | Sim | Sim | Sim |
| **Conversão de Blocos** | Sim | Sim | Sim |
| **Conversão de Blocos Disjuntos** | Sim | Não | Não |
| **Conversão de Atributos Múltiplos** | Sim | Sim | Não |
| **Conversão de Atributos Múltiplos Não Nulos** | Sim | Não | Sim |
| **Conversão de Atributos Identificadores** | Sim | Sim | Sim |
| **Conversão de Atributos de Referência** | Sim | Sim | Sim |

A tabela 1 exibe as diferentes funcionalidades que as regras da seção 3.2, 3.3 e 3.4 formalizam, de forma que ela serve de base para entender conceitualmente as peculiaridades de cada tipo de BD NoSQL na hora de gerar as regras.

## 3.2 Regras de Conversão NoSQL para MongoDB

A primeira ação para se converter para o modelo do MongoDB é analisar o modelo de entrada. Este modelo é definido como um modelo de agregados (LIMA, 2016), do qual é implementado no BrModeloNext como uma árvore, que cada conjunto de agregados é representado por uma hierarquia.

*Figura 24 - Árvore do modelo de agregados*




Fonte: Gerada pelo Autor

Por exemplo, na Figura 24 é representada a árvore a partir do modelo da Figura 19 projetado no BrModeloNext, de forma que é possível ver a disposição dos nodos como modelo de agregados. Cada nodo contém informações se são novos blocos, atributos, únicos entre outros, e suas posições na árvore indicam o nível de agregação em cada bloco. Assim, na Figura 24 é mostrado o *Author* que possui nodos filhos identificados como atributos, no caso *ID_Author* e *Name*, além do nodo *Address* que é um bloco aninhado com mais outros nodos filhos se comportando como atributos.

Com a árvore definida, é possível partir para o próximo passo que se resume em analisar as possibilidades que um modelo de agregados possui e convertê-lo, para documento, no caso deste trabalho para JSON Schema.

Os estados são definidos de acordo com o que o nodo representa. No caso é uma composição se o nodo é uma coleção, um bloco (uma coleção aninhada em outra coleção), um atributo comum, um atributo identificador ou de referência, ou até se o atributo possui disjunção com outro atributo. Unido a esses fatores, é considerada a cardinalidade do nodo que está sendo avaliado, que pode variar em 0-1 a N-N tendo implicações, assim, a forma de representação como um atributo simples, um array, ou

até um array com requisitos mínimos de existência, como é caso de cardinalidades 1-1 ou 1-N visto que é necessário pelo menos um elemento no conjunto para que o mesmo esteja valido.

Caso 1: O nodo é uma coleção. Nesse caso é gerada uma nova coleção, como um novo documento, do qual suas propriedades é definidas a partir dos filhos desse nodo que podem ser novas coleções (blocos nesse caso) ou atributos.

Caso 2: O nodo é um bloco. Nesse caso é verificado se esse bloco possui disjunção com outro bloco, caso verdadeiro, é adicionado primeiro a propriedade oneOf no escopo do documento pai.

Após esse passo é executado algum dos casos 2.1 - 2.4.

Caso 2.1: O bloco é de cardinalidade 0-1. Nesse caso é gerado um novo documento dentro da coleção pai, com o seu tipo sendo object**.**

Caso 2.2: O bloco é de cardinalidade 1-1. Nesse caso é gerado um novo documento dentro da coleção pai, com seu tipo sendo object além de adicionar esse novo bloco ao conjunto de requeridos, o qual é o conjunto que contem objetos que não podem ser vazios, do documento pai.

Caso 2.3: O bloco é de cardinalidade 0-N: Nesse caso é gerado um novo documento com o tipo array, que possui atributo items. O item recebera a definição de que tipo de dados é esperado receber dentro desse array ou conjunto. Como o nodo é do tipo bloco esperamos receber vários blocos, portanto itens é definido como tipo Object**.** Um exemplo dessa regra pode ser visto na Figura 25 já no formato de JSON Schema, onde o array *endereços* espera 0-N documentos do tipo Object.

*Figura 25 - Exemplo Caso 2.3 Mongo*

```
17   enderecos: {
18     bsonType: "array",
19     minItems:0,
20     items:[
21     {
22       bsonType: "object",
23       properties:{
24         rua: { bsonType: "string"},
25       },
26       additionalProperties : false,
27     }
28   ],
29   },
```

Fonte: Gerada pelo Autor

Caso 2.4: O bloco é de cardinalidade 1-N. Nesse caso é gerado um novo documento como detalhado no caso 2.3 com a diferença que é definido o número mínimo de itens pela cardinalidade mínima escolhida, além de adicionar o bloco a conjunto de requeridos do documento pai.

Caso 3. O nodo é um atributo

Caso 3.1 O atributo é identificador. Nesse caso é criado um atributo do tipo Objectid dentro da coleção ou bloco pai, além de adicionar o atributo ao conjunto de requeridos do documento pai.

Caso 3.2 O atributo é de referência.

Caso 3.2.1 O atributo tem cardinalidade 0-1. Nesse caso é criado um atributo do tipo Objectid dentro da coleção ou bloco pai.

Caso 3.2.2 O atributo tem cardinalidade 1-1. Nesse caso é criado um atributo do tipo Objectid dentro da coleção ou bloco pai, além de adicionar o atributo ao conjunto de requeridos.

Caso 3.2.3 O atributo tem cardinalidade 0-N. Nesse caso é criado um novo documento do tipo array, com a atributos items do tipo Objectid.

Caso 3.2.4 O atributo tem cardinalidade 1-N. Nesse caso é criado o documento como no caso 3.2.3, além de adicionar o mesmo ao conjunto de requeridos do documento pai.

Caso 3.3 O atributo é comum.

Caso 3.3.1 O atributo tem cardinalidade 0-1. Nesse caso é criado um atributo do tipo especificado no nodo, do qual se enquadra entre os tipos disponíveis para JSON Schema listados na seção 2.7.2

Caso 3.3.2 O atributo tem cardinalidade 1-1. Nesse caso é criado um atributo como no Caso 3.3.1, além de adicionar o atributo ao conjunto de requeridos do documento pai.

Caso 3.3.3 O atributo tem cardinalidade 0-N. Nesse caso é criado uma array, com o atributo items sendo do tipo especificado no nodo.

Caso 3.3.4 O atributo tem cardinalidade 1-N. Nesse caso é criado um atributo como no caso 3.3.3, além de adicionar o atributo ao conjunto de requeridos do documento pai.

Apesar de algumas simplificações para facilitar a formalização das regras e especificidades da notação JSON Schema, essas regras cobrem todos os casos necessário para uma correta conversão de NoSQL para JSON Schema de forma que se possa portar para MongoDB. Assim os algoritmos mostram partes principais da aplicação dessas regras no processo de conversão.

**Algorithm 1:** Operação Principal do conversor para Mongodb que percorre os primeiros nodos da arvore.

**input :** 1 - Tree with Cells Object Data;
**output:** String with Instructions

```
1 Function convertModeling(ObjectCell)
2 instructions ⟵ ∅;
3 for children in Objectcell do
4 |   if children is a Collection then
5 |   |   instructions ⟵instructions + addCollection(Objectcell)
6 end
7 return instructions
```

**Algorithm 2:** Operação do conversor para Mongodb que percorre todos os filhos da arvore chamando a função de verificação de acordo com seu tipo.

**input :** 1 - A node of the tree; Data; 2-String with Instructions
**output:** String with Instructions

```
1 Function convertModeling(ObjectCell, Instruction)
2 for children in Objectcell do
3 |   if children is a Collection then
4 |   |   newCollection = objectCel[i].getValue();
  |   |    instructions ⟵instructions +
  |   |    checkCollectioncardinalitiesCases(newCollection, instructions)
5 |   else if children is a NoSqlAttributeObject then
6 |   |   newNoSQLAtribute = objectCel[i].getValue();
  |   |    instructions ⟵instructions +
  |   |    checkNOSQLAttributeCardinalitiesCases(newNoSQLAtribute,
  |   |    instructions)
7 |   end
8 end
9 return instructions
```

O Algortimo I representa o início do processo de conversão, percorrendo todos os primeiros nodos da árvore e chamando o método *addCollection* que tem o papel de montar instruções de coleção para esses nodos e chamar métodos para percorrer cada filho dessas coleções. Um desses métodos é o Algortimo 2, que percorre os filhos classificando em coleção ou atributo NoSQL. Para cada caso existe um verificador que é apresentado no, algoritmo, 3 e 4.

**Algorithm 3:** Operação do conversor para Mongodb verifica em que caso de cardinalidade as Coleções se encaixam.

**input :** 1 - A node Collection; Data; 2-String with Instructions
**output:** String with Instructions

```
1  Function checkCollectioncardinalitiesCases(nodeCollection, Instruction)
2  minimumCardinality ⟵ nodeCollection.getMinimumCardinality();
3  maximumCardinality ⟵ nodeCollection.getMaximumCardinality();
4  if minimum == '1' maximum == '1' then
5  |  instructions ⟵instructions + addBlock(nodeCollection);
6  else if minimum == '1' maximum != '1' maximum != 'n' then
7  |  instructions ⟵instructions + addBlockWithArray(nodeCollection);
8  end
9  else if minimum == '0' maximum == '1' then
10 |  instructions ⟵instructions + addBlock(nodeCollection)
11 end
12 else if minimum == '0' maximum != '1' maximum != 'n' then
13 |  instructions ⟵instructions + addBlockWithArray(nodeCollection);
14 end
15 else if (minimum == '1' maximum == 'n') —— (minimum == '0'
     maximum == 'n') then
16 |  instructions ⟵instructions + addBlockWithArray(nodeCollection);
17 end
18 if isLastChild(nodeColletion) then
19 |  checkForRequiredObjects(nodeColletion);
20 return instructions
```

O algortimo 3 tem o papel de tratar os casos das coleções, visto que o mesmo verifica o caso das coleções, chamar o metodo correto para construções das instruções dos casos e, no final, verificar quais nodos devem pertencer à lista de requeridos.

**Algorithm 4:** Operação do conversor para Mongodb verifica em que caso de cardinalidade as Coleções se encaixam.

**input :** 1 - A node Collection; Data; 2-String with Instructions
**output:** String with Instructions

```
1  Function checkNOSQLAttributeCardinalitiesCases(nodeAttribute,
     Instruction)
2  minimumCardinality ⟵ nodeAttribute.getMinimumCardinality();
3  maximumCardinality ⟵ nodeAttribute.getMaximumCardinality();
4  if nodeAttribute.isReferenceAttribute() ——
     nodeAttribute.isIdentifierAttribute() then
5  |  addToRequiredList();
6  if minimum == '1'  maximum == '1' then
7  |  instructions ⟵instructions + addAttribute(nodeAttribute);
8  else if minimum == '1'  maximum != '1'  maximum != 'n' then
9  |  instructions ⟵instructions +
   |    addAttributeWithArray(nodeAttribute);
10 end
11 else if minimum == '0'  maximum == '1' then
12 |  instructions ⟵instructions + addAttribute(nodeAttribute)
13 end
14 else if minimum == '0'  maximum != '1'  maximum != 'n' then
15 |  instructions ⟵instructions +
   |    addAttributeWithArray(nodeAttribute);
16 end
17 else if (minimum == '1'  maximum == 'n') —— (minimum == '0'
     maximum == 'n') then
18 |  instructions ⟵instructions +
   |    addAttributeWithArray(nodeAttribute);
19 end
20 if isLastChild(nodeAttribute) then
21 |  checkForRequiredObjects(nodeAttribute);
22 return instructions
```

O algoritmo 4 tem o mesmo papel do que o 3, com o diferencial que é especifico para atributos NoSQL com algumas verificações a mais, como para atributos idenficadores e referenciais.

## 3.3 Regras de Conversão NoSQL para Cassandra

Para montar as regras de conversão para Cassandra é preciso ter disponível a árvore descrita na seção 3.2, além das especificidades impostas pelo modelo colunar,

de forma que para cada nodo pertencente a arvore, é observado em qual dos casos ele se encaixa.

Essas especificidades são o fato de não conseguir tratar disjunção em um modelo colunar, não restringindo a inserção simultânea de dois blocos com disjunção. Outro detalhe é a forma de representação de relações entre as coleções, que seguem o formato proposto no documento, visto que coleções aninhadas são definidas um novo tipo no Cassandra, que analogamente seriam objetos, e para coleções não aninhas que possuem relações, são usadas chaves primárias e estrangeiras para manter suas relações. Assim definido, a seguir estão as regras usadas para converter o modelo NoSQL para Cassandra.

Caso 1: É uma coleção. Nesse caso o conversor deve gerar uma nova tabela na linguagem CQL.

Caso 2: O nodo é um bloco. Nesse caso, significa que a coleção esta aninhada dentro de uma coleção pai. Assim, o conversor deve gerar um novo tipo que será como uma supercoluna no modelo colunar**.**

Caso 3: Caso seja um atributo identificador. Nesse caso é gerado o atributo dentro da tabela da coleção pai, atribuindo o tipo UUID ao objeto além de adicionar a keyword PRIMARY KEY referenciado o atributo identificador.

Caso 4: Caso seja um atributo de referência, que se refere a uma coleção.

4.1: O atributo tem única cardinalidade (0-1 ou 1-1). Nesse caso é adicionado apenas o tipo UUID ao atributo.

4.2: O atributo tem cardinalidade Múltipla (0-N ou N-N). Nesse caso é gerada uma lista que contenha atributos identificadores, com as keywords LIST<UUID>.

Caso 5: Caso seja um atributo comum.

5.1: O atributo tem única cardinalidade (0-1 ou 1-1). Nesse caso é adicionado o atributo com seu referente tipo.

5.2: O atributo tem cardinalidade múltipla (0-N ou N-N).

5.2.1 O atributo é dos tipos primitivos. Nesse caso é gerado uma lista de atributos com esse tipo primitivo, com as keywords LIST<*'TIPO'*>.

5.2.2: O atributo possui um novo tipo gerado a partir das coleções (análogo ao caso 2.3 da seção 3.2). Nesse caso, é gerada uma lista de atributos do

tipo da nova coleção com a keyword frozen, com a seguinte forma LIST<FROZEN <'NOVO TIPO'>>

**Algorithm 5:** Operação de Verificação do Modelo de Agregados para Cassandra

```
input  : 1 - Tree with Cells Object Data;
         2 - Enum with the type of the current collection.
output: String with Instructions
1  Function verifyCellObjects(ObjectCell, CollectionType)
2  instructions ⟵ ∅;
3  cassandraObjectData ⟵ initialization;
4  if ObjectCell has child then
5      for children in Objectcell do
6          if children is a Collection then
7              if children has identifier attribute then
8                  instructions ⟵
                    verifyCellObjects(children, TableType.TABLE);
9              else
10                 cassandraObjectData.addAttribute(
11                 children.name, CassandraTypes.NEWTYPE,
                    hashMultipleCardinalities(children));
12             end
13         else if children is a NoSqlAttribute then
14             cassandraObjectData.addAttribute(.name, children.type,
                hashMultipleCardinalities(children));
15         end
16     end
17 end
18 if ObjectCell is a Collection then
19     cassandraObjectData.setObjectName(ObjectCell.name) if TableType
        is equals TABLE then
20         instructions ⟵
            instructionsBuilder.genTablesInstructions(cassandraObjectData);
21     else
22         instructions ⟵
            instructionsBuilder.genTypeInstructions(cassandraObjectData);
23     end
24 end
25 return instructions
```

Com as regras e as informações contidas em cada nodo é possível fazer a verificação para converter para o modelo do Cassandra, seguindo os passos descritos no algoritmo 2. No algoritmo é recebida a árvore apresentada anteriormente como ObjectCell e um outro parâmetro *TableType* para identificar o tipo de tabela a ser

criada naquele momento de recursão do algoritmo. Ele é executado recursivamente para cada nodo montando um objeto cassandraObjectData para agregar as informações relevantes de cada nodo. Essas informações são transferidas para o objetivo instructionBuild que tem o papel de gerar as instruções em CQL de acordo com as informações contidas no CassandraObjectData.

## 3.4 Regras de Conversão NoSQL para Redis

Para gerar as regras de conversão para o Redis é preciso ter disponível a árvore descrita na seção 3.2 e levar em consideração que o Redis apresenta a forma mais simplória de modelagem, visto que não é possível tratar objetos agregados como objetos inseridos dentro de outros objetos, sim apenas como tipo string ou valores serializados.

Outro ponto é a validação, que é inexistente no banco, de forma que as regras serviram para definir um modelo que é inserido indiretamente em um script lua para se efetuar validações. Para os casos abaixo, todas as coleções estão sendo consideradas como tipo hash do Redis, e atributos como strings. O conjunto de requeridos é abordada nos casos abaixo. É uma lista com todos os atributos ou blocos de uma coleção que devem ser inseridos para que o esquema seja válido.

Caso 1: É uma coleção. Nesse caso é gerado um escopo de condição recebendo o nome da coleção.

Caso 2: É um bloco e tem cardinalidade (1-1 ou 1-N). Nesse caso é adicionado ao conjunto de requeridos da sua coleção pai.

Caso 3: É um atributo identificador. Nesse caso é adicionado ao conjunto de requeridos da sua coleção pai.

Caso 4: É um atributo cardinalidade (1-1 ou 1-N). Nesse caso é adicionado ao conjunto de requeridos da sua coleção pai.

O algoritmo 6 exibe o processo de verificação do modelo de agregados para Redis onde é feita a montagem do objeto com os dados para serem utilizados na criação das instruções para o banco.

**Algorithm 6:** Operação de Verificação do Modelo de Agregados para Redis

**input :** 1 - Tree with Cells Object Data;
**output:** ObjectData to be used in building instructions

```
1  Function verifyCellObjects(ObjectCell)
2  redisObjectData ⟵ initialization;
3  redisObjectData.setObjectName(ObjectCell.getName());
4  if ObjectCell has child then
5  |  for children in Objectcell do
6  |  |  if children is a Collection then
7  |  |  |  if (children.getMinimumCardinality() == 1) then
8  |  |  |  |  redisObjectData.addAttributes(children.getName());
9  |  |  |  else
10 |  |  else if children is a NoSqlAttribute then
11 |  |  |  if (children.getMinimumCardinality() == 1 ——
   |  |  |    attribute.isIdentifierAttribute()) then
12 |  |  |  |  redisObjectData.addAttributes(children.getName());
13 |  |  |  else
14 |  |  end
15 |  end
16 end
17 return redisObjectData
```

Nele é recebido a árvore como ObjectCell e é instanciado um *redisObjectData* com o nome da coleção inicial. Se a coleção tiver filhos, são verificados todos os filhos da coleção e comparados de acordo com os casos 2 a 4, ao passo que no final é retornado o *redisObjectData* com as informações necessárias para classe *rededisInstructionBuilder* montar o script final em lua.

## 3.5 Implementação

### 3.5.1 Ferramentas Utilizadas

Esta seção descreve brevemente as ferramentas e linguagens utilizadas para desenvolvimento do projeto.

#### 3.5.1.1 Java

Java é uma linguagem orientada a objetos criada na década de 90 (ORACLE, 2018) com o propósito de ser portável, robusta, flexível e de fácil aprendizado se comparada a outras linguagens populares na época como C++. A linguagem compila o código para bytecode, que este é interpretado para pela máquina virtual java (JVM), que pode ser executaod em qualquer maquina que possua a JVM, tornando a linguagem muito mais flexível. (ROUSE, 2016)

No presente trabalho a versão utilizada é a versão 8, que apresenta novas funcionalidades como funções lambdas e além de ser a versão utilizada para o desenvolvimento do aplicativo BrModeloNext, sendo assim, essa é a justificativa pela escolha da linguagem para o desenvolvimento das funcionalidades.

#### 3.5.1.2 BrModeloNext

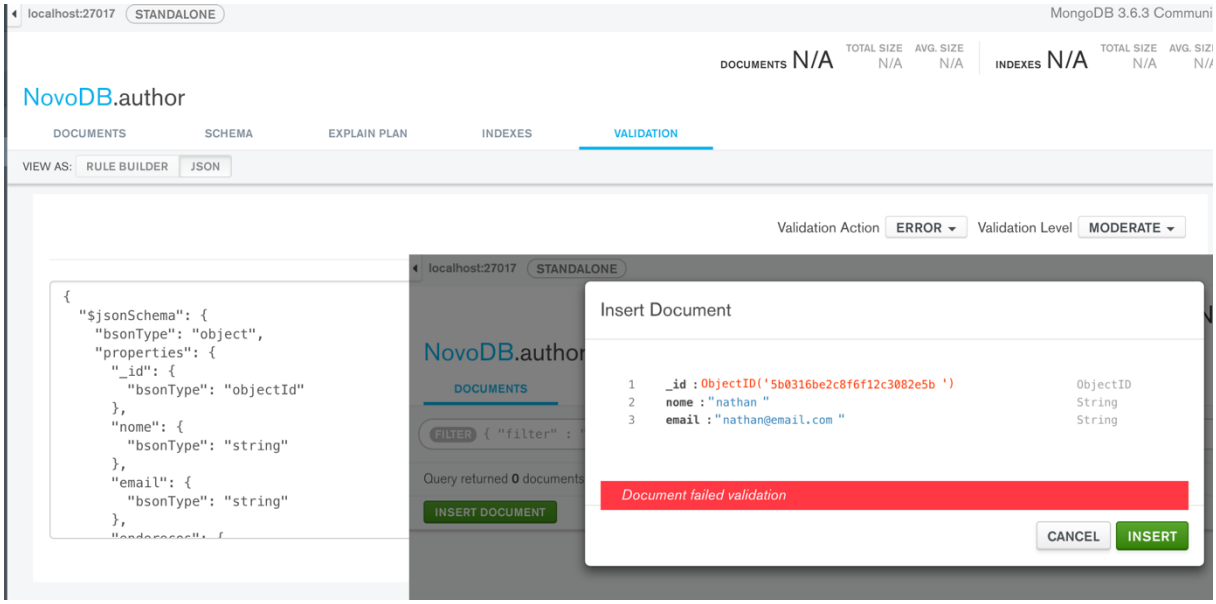
A ferramenta BrModeloNext teve seu funcionamento detalhado na seção 2.6. Sua utilização na implementação foi primeiramente como ponto de partida para criação das novas funcionalidades que são o foco deste trabalho, detalhadas na seção 3.1. Ela também tem seu uso para criar modelos de teste, de conversão conceitual para NoSQL, e validação dos resultados obtidos que são mostrados na seção 4.

### 3.5.1.3 MongoDB Compass

O Mongodb Compass é uma ferramenta gráfica de visualização para MongoDB, que possibilita ver e administrar os dados inseridos no banco, além de possibilitar depurar e otimizar coleções, além de abstrair esquemas.

De mesma forma que é possível manipular os dados do MongoDB apenas pela sua interface shell javascript, o uso do MongoDB Compass é justificado pelos seus recursos de visualização, principalmente no escopo de validações, do qual torna a ver as regras geradas na seção 3.2 muito mais intuitivas e com um feedback mais rápido.

*Figura 26 - Interface Mongodb Compass e Funcionalidades*




Fonte: (Gerada pelo Autor)

A Figura 26 exibe a interface do Compass, a qual mostra as regras inseridas no banco, e a resposta do MongoDB ao tentar inserir um documento que não se enquadra nas mesmas.

### 3.5.1.4 CQLSH

O Cassandra Query Language Shell (CQLSH) é a interface entre o usuário e o Cassandra. Nele é possível administrar todos os dados do banco, além de efetuar buscas e otimizações.

*Figura 27 - Exemplo Select CQLSH*

```
cqlsh:animalkeyspace> select * from monkey ;

 identifier                           | species              | nickname  | population
--------------------------------------+----------------------+-----------+------------
 5132b130-ae79-11e4-ab27-0800200c9a66 |       Capuchin monkey |      cute |     100000
 5132b130-ae79-11e4-ab27-0800200c9a66 | Small Capuchin monkey | very cute |        100
 7132b130-ae79-11e4-ab27-0800200c9a66 |         Rhesus Monkey |  Handsome |     100000

(3 rows)
```

Fonte: (AKHILL, 2015)

Sua escolha foi baseada por ser a versão padrão de comunicação com o Cassandra e além de possuir uma forma de exibição satisfatória para visualização.

### 3.5.1.5 Redis-Cli

O Redis-Cli é a interface que recebe comandos de usuário e os interpreta em ações para o Redis. Nele é possível administrar e visualisar os dados do banco, executa-los diretamente no shell, ou executar arquivos externos com comandos ou scripts lua.

*Figura 28 - Rodando Script lua em Redis-Cli*

```
$ cat /tmp/script.lua
return redis.call('set',KEYS[1],ARGV[1])
$ redis-cli --eval /tmp/script.lua foo , bar
OK
```

fonte: (REDIS, 2017)

Sua escolha é dada por ser a versão padrão que acompanha o Redis e apresentar diversas formas de interação.

### 3.5.1.6 Outras Ferramentas

Outras ferramentas utilizadas que merecem uma breve listagem foram a IDE Eclipse e o editor de texto Sublime, que tiveram seu uso para implementação e edição respectivamente, além do terminal para manipular diretório e arquivos.

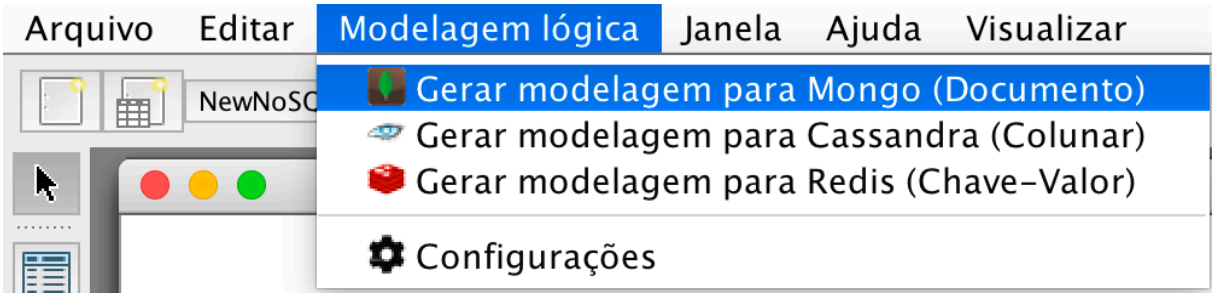
## 3.5.2 Funcionalidades

As funcionalidades foram todas implementadas para a plataforma BrModeloNext com o uso das tecnologias e ferramentas especificas na seção 3.5.1.

### 3.5.2.1 Converter Esquema NoSQL de Agregados para MongoDB

A fim de poder converter um esquema NoSQL em regras específicas para MongoDB é necessário primeiro gerar um esquema NoSQL na ferramenta BrModeloNext. Próximo passo é selecionar a opção de conversão para MongoDB no novo menu de modelagem lógica, exibido na imagem abaixo.

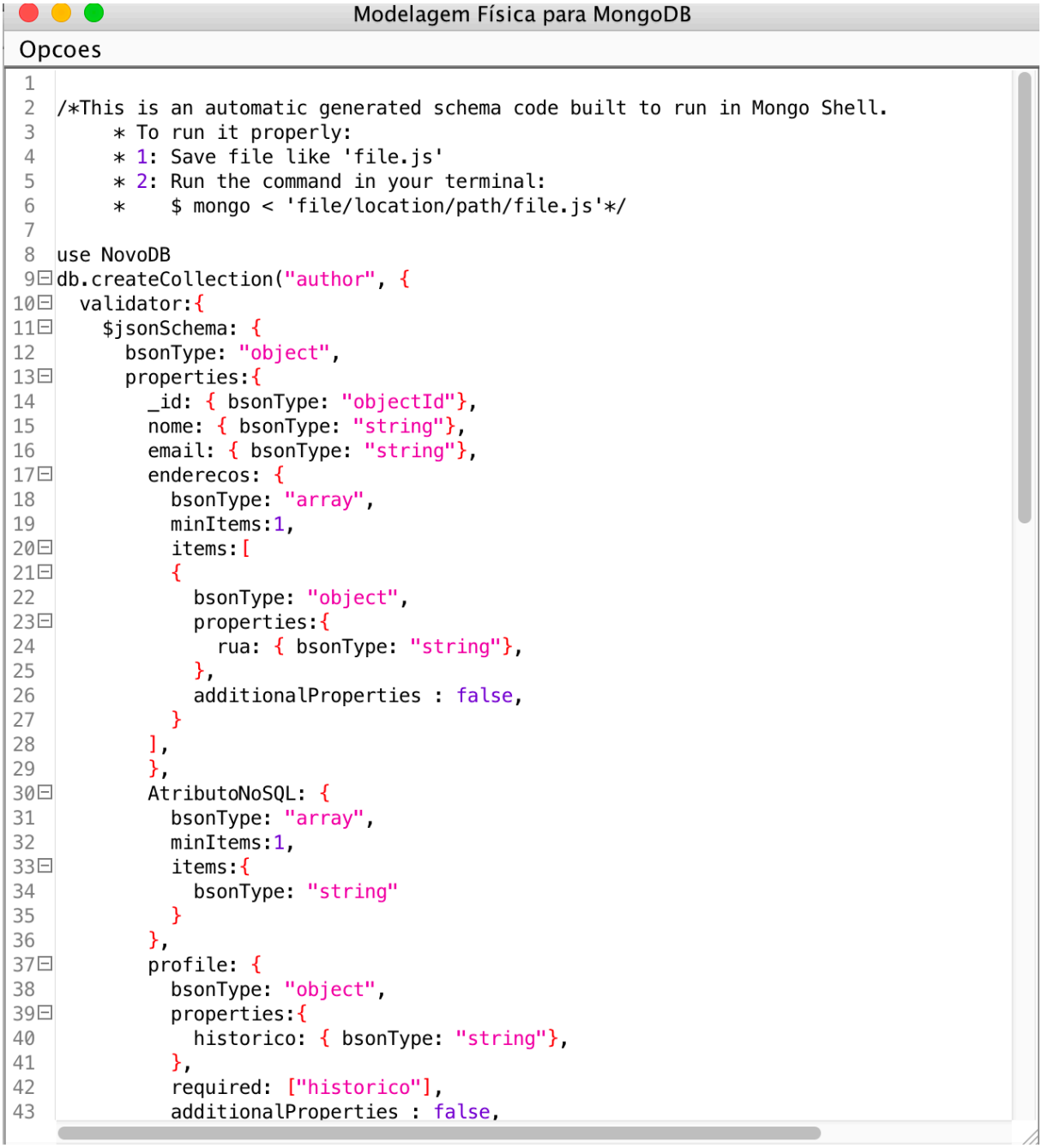
*Figura 29 - Menu de modelagem lógica BrmodeloNext*




Fonte: (Gerada pelo Autor)

Após selecionado, é chamado o método *convertModeling* da Classe LógicalToMongoConversor que analisa o esquema NoSQL fornecido e aplica todas as regras de conversão definidas na seção 3.2.

*Figura 30 - Resultado da Conversão para Mongodb*




```
1
2  /*This is an automatic generated schema code built to run in Mongo Shell.
3       * To run it properly:
4       * 1: Save file like 'file.js'
5       * 2: Run the command in your terminal:
6       *    $ mongo < 'file/location/path/file.js'*/
7
8  use NovoDB
9  db.createCollection("author", {
10   validator:{
11     $jsonSchema: {
12       bsonType: "object",
13       properties:{
14         _id: { bsonType: "objectId"},
15         nome: { bsonType: "string"},
16         email: { bsonType: "string"},
17         enderecos: {
18           bsonType: "array",
19           minItems:1,
20           items:[
21           {
22             bsonType: "object",
23             properties:{
24               rua: { bsonType: "string"},
25             },
26             additionalProperties : false,
27           }
28         ],
29         },
30         AtributoNoSQL: {
31           bsonType: "array",
32           minItems:1,
33           items:{
34             bsonType: "string"
35           }
36         },
37         profile: {
38           bsonType: "object",
39           properties:{
40             historico: { bsonType: "string"},
41           },
42           required: ["historico"],
43           additionalProperties : false,
```

Fonte: (Gerado pelo Autor)

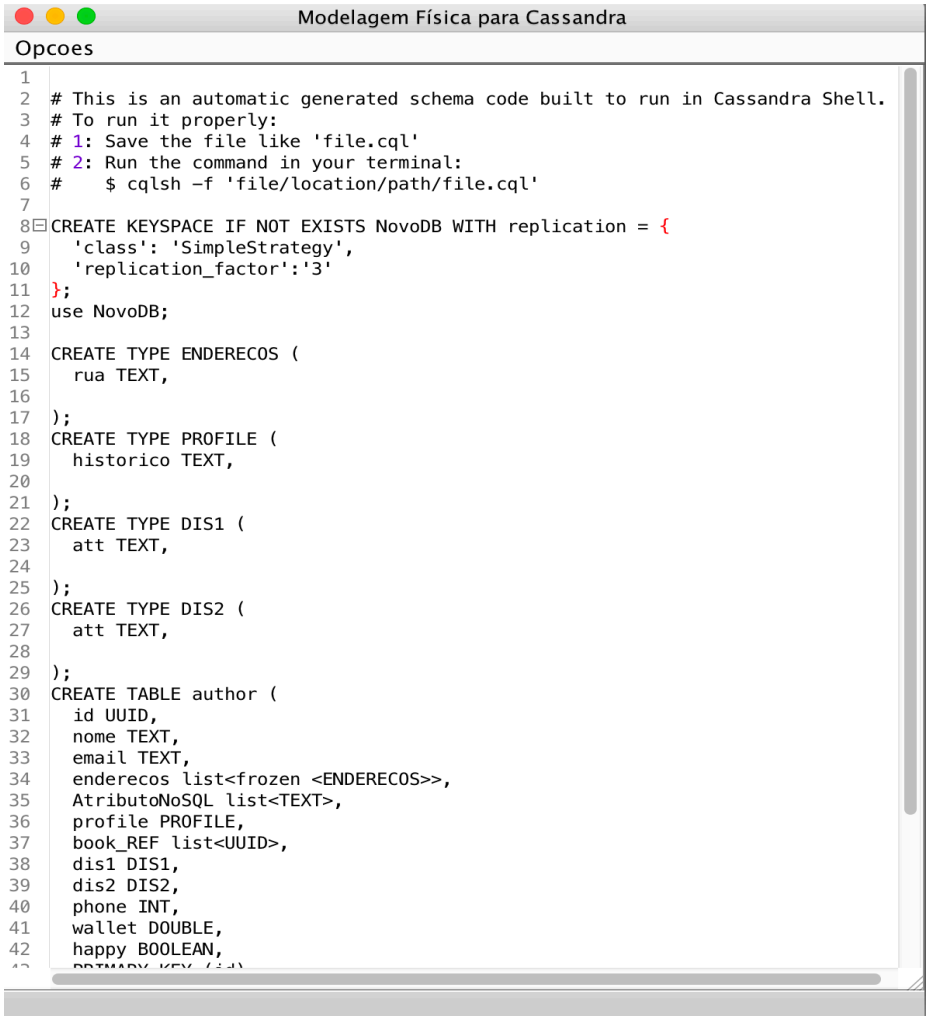
O resultado da aplicação das regras é mostrado na Figura 30, visto que é criada uma nova janela com o código para ser aplicado para MongoDB com suas respectivas instruções.

### 3.5.2.2 Converter Esquema NoSQL de Agregados para Cassandra

O processo de conversão de esquema NoSQL de agregados para Cassandra envolve definir previamente um modelo NoSQL e selecionar a opção de conversão para Cassandra, como exibido na Figura 29.

Essa ação invoca o método *convertModeling* da classe LogicalToCassandra conversor. Essa classe necessita de outras 3 classes auxiliares para poder converter, que são *CassandraInstructionBuilder,* responsável por lidar com toda a montagem dos tokens específicos para CQL, além das classes *CassandraObjectData* e *CassandraAttribute* que definem objetos de dados para possibilitar a classe *CassandraInstructionBuilder* montar de forma correta as regras finais.

*Figura 31 - Resultado de Conversão para Cassandra*




Fonte: Gerada pelo Autor

Após o processo de conversão é gerado o resultado final mostrado na Figura 31 com a janela e o código em CQL e seus respectivas instruções para serem aplicadas.

#### 3.5.2.3 Converter Esquema NoSQL de Agregados para Redis

O processo de conversão de esquema NoSQL de agregados para Redis envolve definir previamente um modelo NoSQL e selecionar a opção de conversão para Redis, como exibido na Figura 29. Essa ação invoca o método *convertModeling* da classe LogicalToRedisConversor. Para seu funcionamento a classe necessita de outras duas classes auxiliares, a RedisObjectData e RedisInstructionBuilder. A primeira é responsável por gerar objeto de dados para o formato das regras do Redis, como visto no algoritmo 6. A outra tem o papel de receber esses dados e gerar a instruções Lua a partir deles.

A figura 32 mostra o resultado final do processo de conversão, de forma que é possível observar as coleções sendo definidas dentro da função *checkColectionCases* com seus respectivos atributos, a fim de serem validados na execução do código.

*Figura 32 - Resultado Conversão Redis*

Opcoes

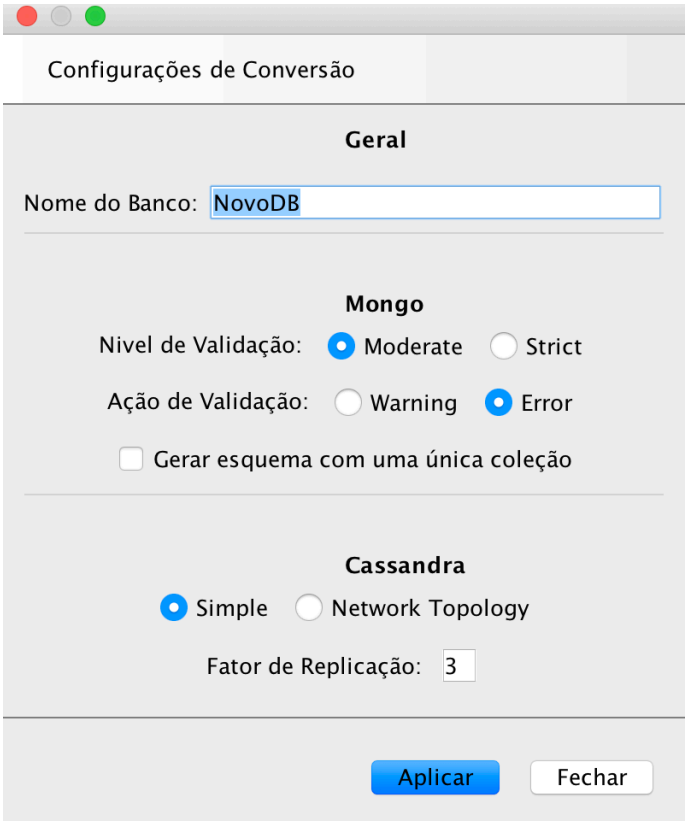
```
46          return t
47  end
48
49  local function findCollection(colToInsert)
50       local colName = split(colToInsert, ":")[1]
51       return colName
52  end
53
54  local function CheckRequiredAttributes(colection_required, length)
55       local isCorrect = true
56
57       if (table.getn(KEYS) - 1  < length) then
58            return false
59       end
60
61       for _,key in ipairs(KEYS) do
62            if (_ > 1) then
63                 if (not colection_required[key]) then
64                      isCorrect = false
65                 end
66            end
67       end
68
69       return isCorrect
70  end
71
72  local function checkCollectionCase (colName)
73    if (colName == 'author') then
74      local required = {id = true,  email = true,  enderecos = true,  AtributoNoSQL = true,  profile =
75      return CheckRequiredAttributes(required, 7)
76    elseif (colName == 'book') then
77      local required = {name = true,  id = true,  author_REF = true,  ISBN = true,  Publisher_REF = tru
78      return CheckRequiredAttributes(required, 5)
79    elseif (colName == 'Publisher') then
80      local required = {id = true,  name = true,  location = true,  }
81      return CheckRequiredAttributes(required, 3)
82    end
83  end
84
85  return checkCollectionCase(findCollection(KEYS[1]));
```

Fonte: Gerado Pelo Autor

### 3.5.2.4 Configurar Opções de Conversão

Como o objetivo da conversão é tornar o processo de criação e definição dos modelos para os bancos de dados algo intuitivo e prático, é interessante que o usuário possa customizar algumas ações de conversão, de forma que não tenha que altera-las em outro momento depois que o código tenha sido gerado.

*Figura 33 – Configurações de Conversão*




Fonte: (Gerado pelo Autor)

Assim, foi criado um menu de opções que definem nome do banco que vai ser gerado, o nível de validação e a ação de resposta do MongoDB quando as regas são violadas. Além disso, se o usuário preferir, é possível criar todas as coleções como uma única coleção, forçando a modelagem do banco ser um único grande documento com blocos aninhados.

Na parte do Cassandra foi definida configurações iniciais como o tipo replicação e o fator desejado, para que não se tenha que alterar no futuro.

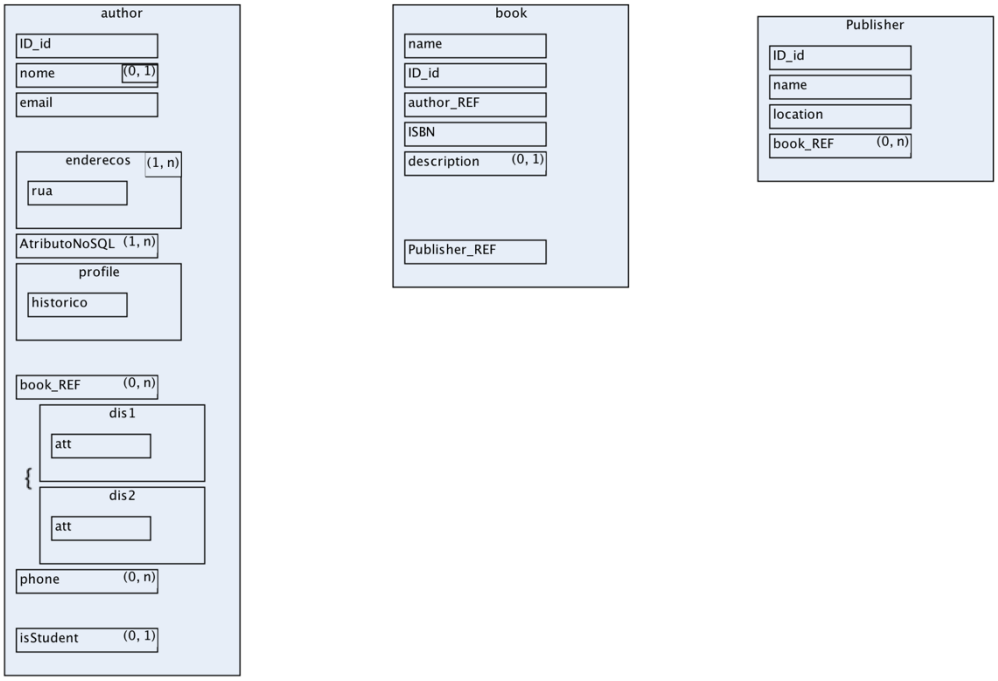
Os relacionamentos dessas funcionalidades com as classes de conversão podem ser vistos na arquitetura da seção 3.1.4

# 4 Resultado Obtidos

Este capítulo apresenta os resultados obtidos com as novas funcionalidades descritas na seção 3 e implementadas na ferramenta BrModeloNext. A avaliação leva em conta se a conversão é correta para todos os casos propostos e como as regras se comportam quando inseridas nos seus respectivos bancos de dados.

Os testes e avaliações utilizaram o caso da Figura 43 modelado na ferramenta BrModeloNext. Esse modelo é composto de autores, livros e editoras, dos quais cada autor tem relação com livros e livros tem relações com autores e editoras. Os atributos foram pensados a fim de expor todos os casos relatados na seção 3 e testar a qualidade das conversões.

*Figura 34 - Modelo NoSQL para Testes de Validação*




Fonte: (Gerado pelo Autor)

## 4.1 Teste do modelo de caso no MongoDB

O teste para MongoDB utilizou o modelo da Figura 34, que resultou no código para MongoDB descrito no apêndice. O código gerado foi salvo e executado como arquivo de entrada para MongoDB a partir das instruções descritas no início do arquivo.

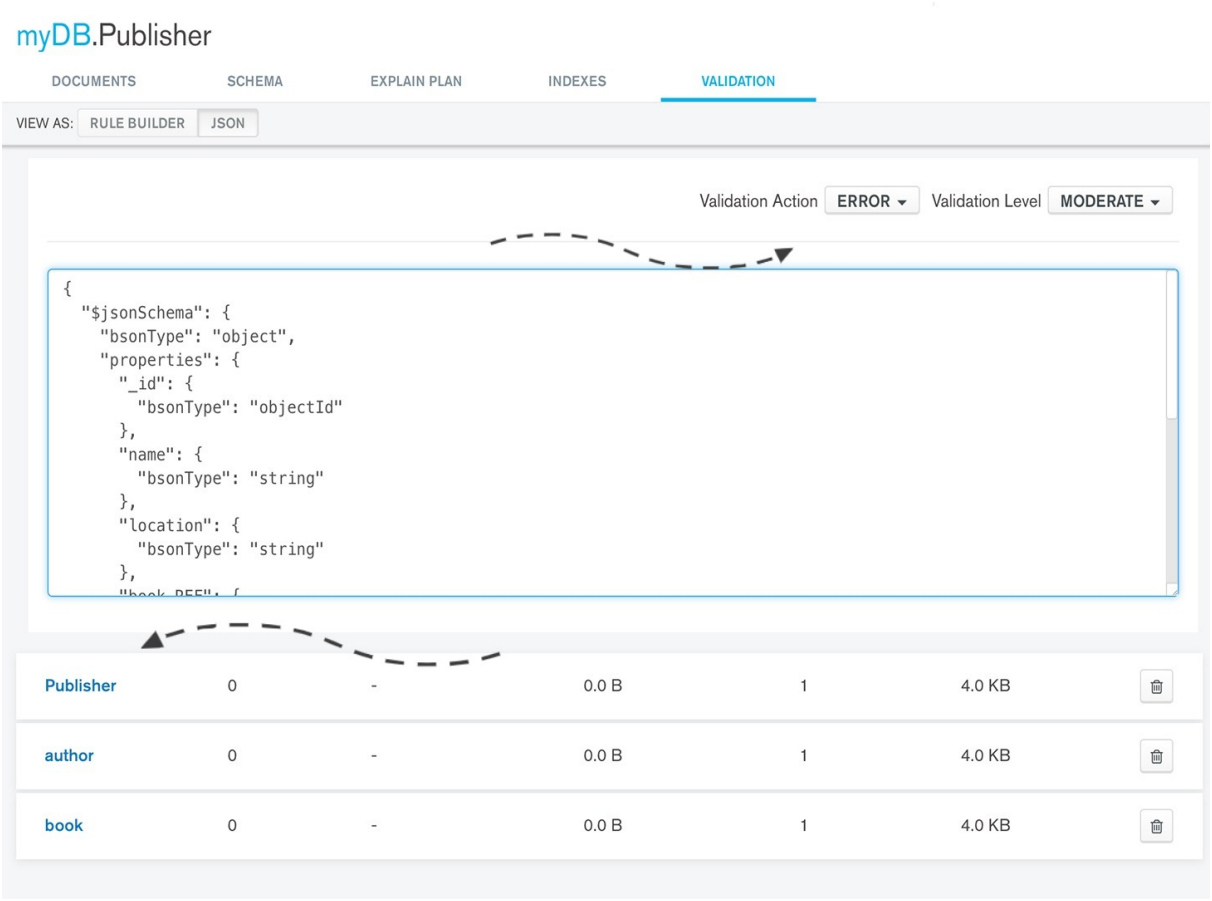
*Figura 35 - Execução Script com Sucesso no MongoDB*


Fonte: (Gerado pelo Autor)

Na Figura 35 é possível ver o script sendo executado a partir do shell no MongoDB, visto que cada “ok” sinaliza que as regras de cada coleção foram inseridas com sucesso.

Com as regras inseridas, o próximo passo é ver como elas estão no banco, para isso foi utilizado o Mongodb Compass.

*Figura 36 - Resultado da Inserção no Compass*

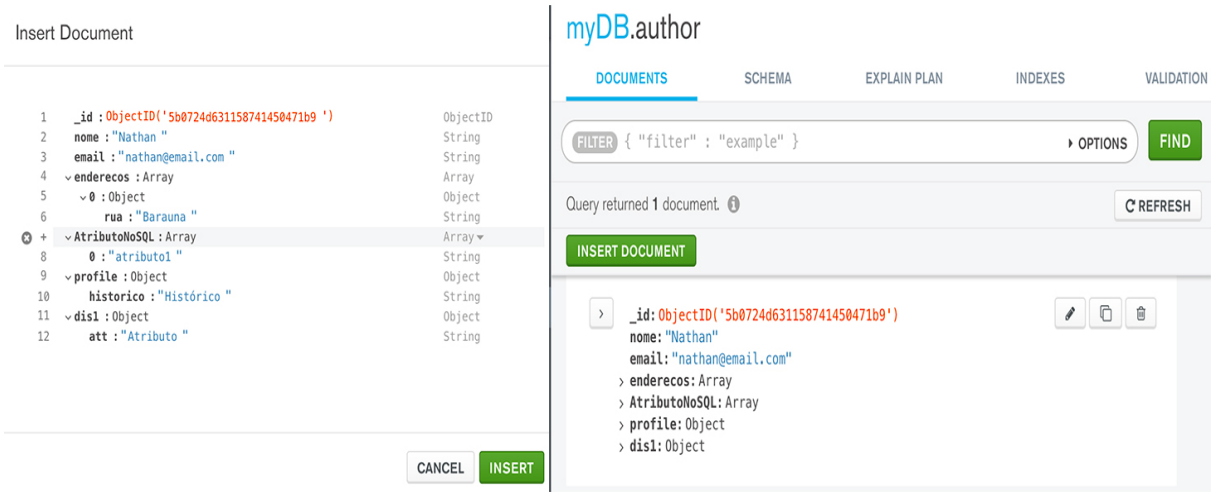



Fonte: (Gerado pelo Autor)

A Figura 36 na parte mais abaixo mostra que todas as 3 coleções foram criadas com sucesso e, em cima, é possível observar que a forma de validação foi registrada como o esperado. O último passo para validar esse caso é fazer testes de inserção de documentos JSON corretos e incorretos, ou seja, documentos que estão de acordo com as regras propostas, do qual é esperado que insira corretamente no banco e documentos que não seguem as regras de forma que o esperado é a não inserção junto com a mensagem de erro.

O caso de sucesso é exibido na Figura 37. A esquerda da figura inserindo os dados do Author, com todos seus atributos e com seus respectivos tipos, e à direita o documento inserido no banco.

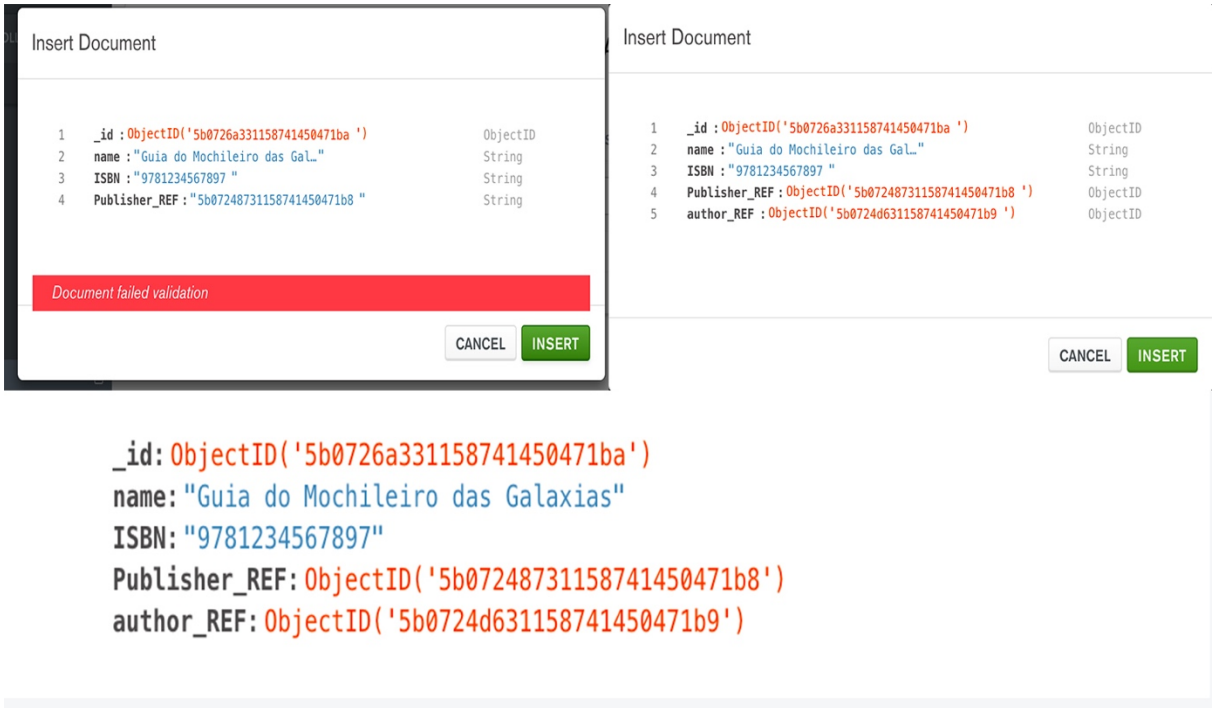
*Figura 37 - Inserção com sucesso MongoDB*




Fonte: (Gerado pelo Autor)

Em contraponto é exibido a esquerda da Figura 38 o caso de erro, ao passo que é feita a tentativa de inserir um documento na coleção livro, porém sem adicionar uma referência ao autor de forma que é requerida e considerar o tipo dos identificadores de referência como *String* e não como tipo *ObjectId* como pede o esquema gerado. Já nas outras partes da imagem é possível ver a forma correta e o documento inserido no banco.

*Figura 38 - Inserção com erro MongoDB*




Fonte: (Gerado pelo Autor)

O próximo passo a ser testado é a geração de uma coleção única com todas as outras coleções do modelo criado sendo agregadas nessa única coleção. Para executar esse modo é necessário selecionar a opção “Gerar esquema com uma única coleção”, exibida na Figura 33. Após esse passo é selecionado o modo de conversão para MongoDB e é gerado o código mostrado na Figura 38, que pode ser visto com mais detalhes no apêndice.

*Figura 39 - Teste de Coleção Única Mongodb*

Modelagem Física para MongoDB

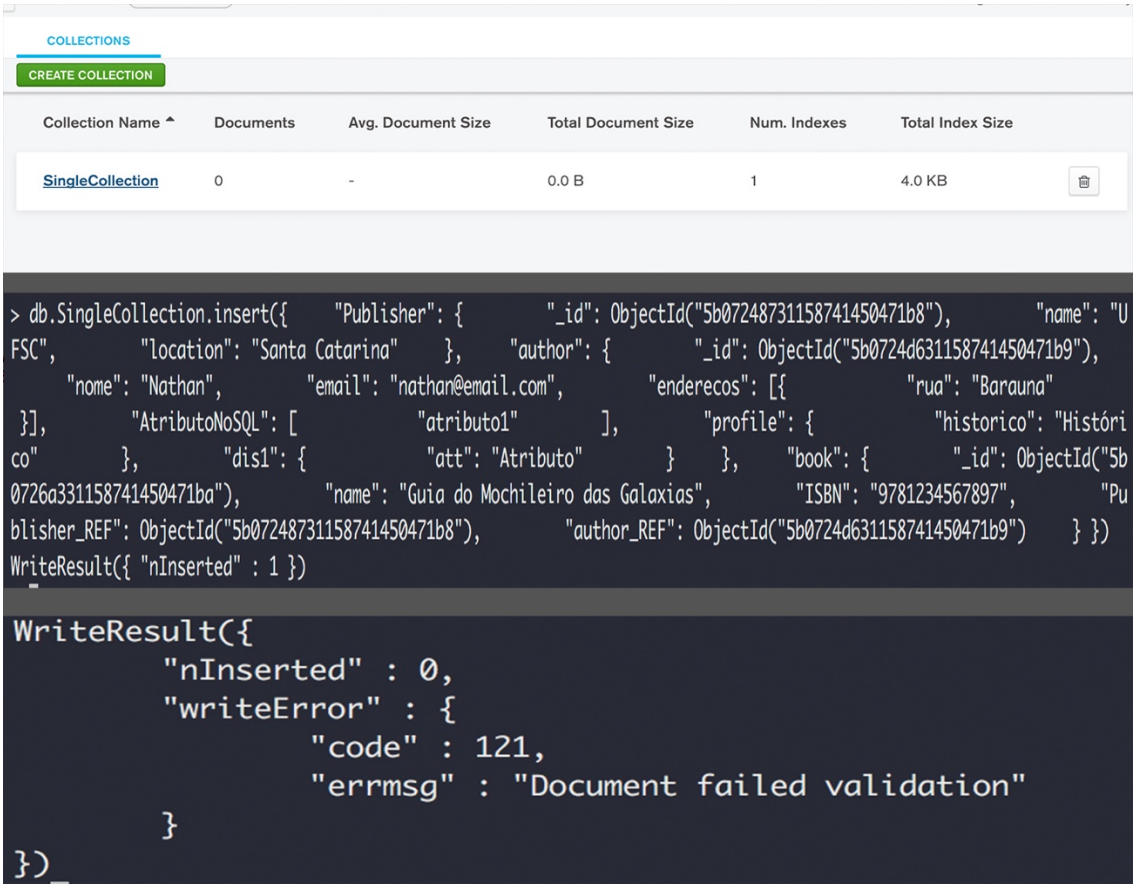
Opcoes

```
validator:{
  $jsonSchema: {
    bsonType: "object",
    properties:{
      author: {
        bsonType: "object",
        properties:{
          _id: { bsonType: "objectId"},
          nome: { bsonType: "string"},
          email: { bsonType: "string"},
          enderecos: {
            bsonType: "array",
            minItems:1,
            items:[
            {
              bsonType: "object",
              properties:{
                rua: { bsonType: "string"},
              },
              additionalProperties : false,
            }
          ],
          },
          AtributoNoSQL: {
            bsonType: "array",
            minItems:1,
            items:{
              bsonType: "string"
            }
          },
          profile: {
            bsonType: "object",
            properties:{
              historico: { bsonType: "string"},
            },
            required: ["historico"],
            additionalProperties : false,
          },
          book_REF: {
            bsonType: "array",
            minItems:0,
            items:{
              bsonType: "objectId"
```

Fonte: (Gerado pelo Autor)

Esse código foi salvo e executado com sucesso de forma análoga ao código anterior. A seguir foi feito o teste de sucesso e o de falha, os quais foram feitos utilizando o shell do MongoDB por ser mais conveniente ao trabalhar com documentos maiores.

*Figura 40 - Resultados Testes de Coleção Única*




Fonte: (Gerado pelo Autor)

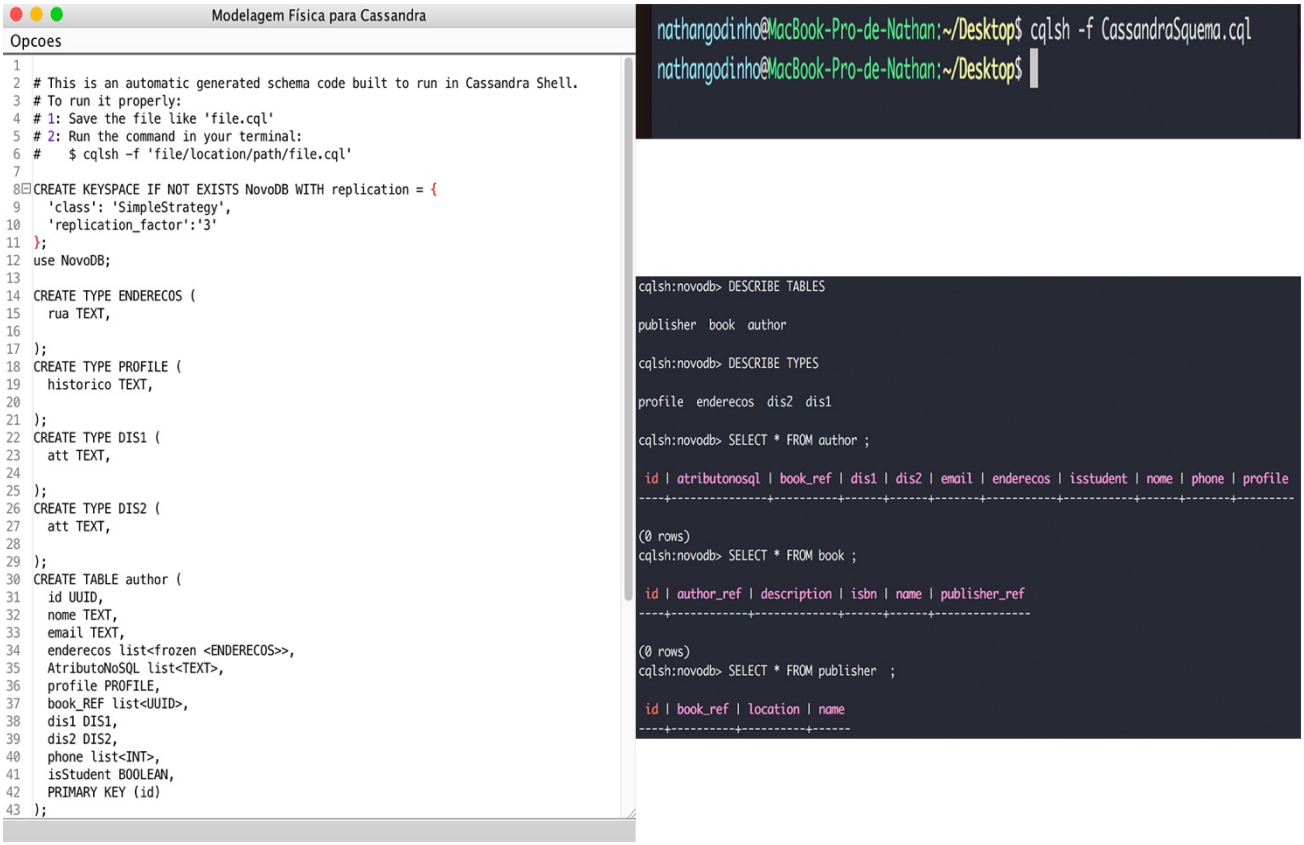
A Figura 38 exibe a coleção inserida, o teste de sucesso executado com o documento do anexo 3. E por fim o teste de erro a partir do documento do anexo 4.

## 4.2 Teste para Cassandra

Os testes para o Cassandra foram baseados no modelo da Figura 33. A partir desse modelo foi gerado o código convertido de forma análoga ao descrito na seção 4.1. Parte do código gerado é exibido na Figura 41, de forma que o restante pode ser

consultado no apêndice. A figura também apresenta execução com sucesso do código no shell cql e a visualização das tabelas criadas de forma esperada.

*Figura 41 - Teste do Código CQL*




Fonte: (Gerado pelo Autor)

O próximo passo é o teste de inserção no banco para ver se o modelo e as regras estão se comportando como o esperado. Assim foram criados os comando de inserção descritos no anexo 6 e feito o teste.

*Figura 42 - Caso de falha na inserção*


Fonte: (Gerado pelo Autor)

A Figura 42 mostra o primeiro caso de falha, no qual foi feita uma tentativa de inserir um valor do tipo *String* de forma que o esperado era uma lista. O resultado foi a não inserção como esperado.

*Figura 43 - Caso de Sucesso Inserção no Cassandra*


Fonte: (Gerado pelo Autor)

Figura 43 mostra a inserção com sucesso no banco conforme esperado pelo modelo.

## 4.3 Teste para Redis

Os testes para o Redis foram baseados no modelo da Figura 33. A partir desse modelo foi gerado o código convertido de forma análoga ao descrito na seção 4.1 e 4.2. O código gerado pode ser visto na Figura 32 e consultado no apêndice deste trabalho.

A forma de execução do código se diferencia da forma como foi feita para os bancos anteriores, visto que o código gerado eram instruções para os comandos ou funcionalidade predefinidas nos bancos. Neste caso para o Redis, o código gera instruções da linguagem de programa Lua que devem ser interpretadas no ambiente do Redis. Para tal é necessário definir padrões para que o teste ocorra de forma correta.

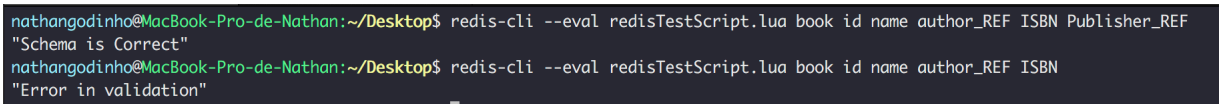
O padrão de execução é que para cada coleção que se queira testar, executar o comando na seguinte forma:

```
$ redis-cli --eval file.lua' collectionToTest:id attribute1
attribute2...atributeN
```

O código acima chama o cliente do Redis com o indicador *–eval* para interpretar um arquivo Lua. As outras palavras-chaves são parâmetros de entrada para o código que vai ser interpretado. Por definição, o primeiro parâmetro deve ser o nome da coleção que se quer inserir. Por exemplo o hash author:1. Nos parâmetros seguintes são os atributos que se quer inserir. Por exemplo nome, email, endereço.

Seguindo o exemplo anterior, o código gerado foi salvo e executado para o caso de sucesso primeiramente e, posteriormente, para o caso de falha, como pode ser visto na Figura 44

*Figura 44 - Teste para Redis*




Fonte: (Gerado pelo Autor)

# 5 Conclusão

O presente trabalho de conclusão de curso teve como seu principal objetivo desenvolver regras de mapeamento para as diferentes classes de bancos de dados NoSQL.

O principal encorajamento para este trabalho é a possibilidade de poder prover uma ferramenta completa, de forma que desenvolvedores e usuários possam modelar esquemas em alto nível e ter o resultado dessa modelagem pronta para ser aplicada no banco NoSQL de sua escolha, seja ele MongoDB, Cassandra ou Redis. Além disso, apesar das implementações serem específicas para os bancos descritos previamente, as regras desenvolvidas nesse trabalho servem de base para serem aplicadas e outros bancos, expandindo assim a riqueza da ferramenta BrModeloNext.

Para atingir estes objetivos foi feito uma pesquisa sobre trabalhos relacionados envolvendo ferramentas de conversão de modelos de agregados para NoSQL, dos quais não foram encontrados nenhuma referência relevante. Também foram levantadas toda a base teórica para desenvolver as regras, que culminou em entender o modelo de agregados, os diferentes tipos de bancos, as especificidades de cada banco trabalhado, tecnologias e a ferramenta BrModeloNext.

Os objetivos descritos na seção 1.4 foram concluídos como esperado, porém com ressalvas a implementação do Redis que possui características que limitam a sua modelagem, o que torna a solução para este banco mais simples e com uma menor aplicação prática, diferente das soluções apresentadas para MongoDB e Cassandra. Todavia, conforme constatado na seção 4, os testes foram bem-sucedidos e os objetivos do projeto podem ser considerados como atingidos.

O código fonte e as instruções para uso do mesmo, estão com acesso livre e disponíveis no repositório em (REUTER, 2018).

Pensando em trabalhos futuros, a primeira ideia seria melhorar a funcionalidade de validação do Redis, para que possa além de verificar se um dado modelo é válido, também inseri-lo no banco, caso seja mesmo correto. Outro ponto a ser abordado é melhorar a ferramenta BrModeloNext, a fim de determinar e avisar o usuário, se um dado modelo, não se encaixa para algum dos possíveis tipos de BD's NoSQL abordados neste trabalho.

# Referências


1KEYDATA. **Data Modeling Levels**. 1keydata, 1 nov. 2017. Disponivel em: <https://www.1keydata.com/datawarehousing/data-modeling-levels.html>.

AKHILL, **A Practical Introduction To Cassandra Query Language**, A Bias for Action, 2 abr, 2015. Disponivel em: < http://abiasforaction.net/a-practical-introduction-to-cassandra-query-language/>. Acesso em: 20 ago. 2017.

ALMEIDA, C. **Uma Pequena Introdução a Big Data**. Concrete, 2017 jun. 2. Disponivel em: <https://www.concrete.com.br/2017/06/02/uma-pequena-introducao-a-big-data/>. Acesso em: 2017 ago. 2017.

ALMEIDA, P. M. B. **Noções básicas sobre metodologia de pesquisa científica**. Mba.eci. Disponivel em: <http://mba.eci.ufmg.br/downloads/metodologia.pdf>. Acesso em: 1 jan. 2017.

BANKER, K. **Mongodb in Action,** Shelter Island, USA: Manning Publications Co, 2012. ISBN 9781935182870

BUGIOTTI, F**. A logical approach to NoSQL databases**. Cabibbo.dia.uniroma3.it, jun. 2014. Disponivel em: <http://cabibbo.dia.uniroma3.it/pub/noam.pdf>. Acesso em: 20 nov. 2017.

CÂNDIDO, C. H. **BrModelo: Ferramenta de Modelagem Conceitual de Banco de Dados**. Sis4. Disponivel em: <http://sis4.com/brModelo/monografia/monografia.htm#_Toc102163870>.

CARLSON, J. L. **Redis in Action**. Manning Publications Co., 2013. ISBN 9781935182054.

CASSANDRA. **Cassandra Data Model Rules**. Guru99. Disponivel em: <https://www.guru99.com/cassandra-data-model-rules.html>. Acesso em: 12 jan. 2017.

COUGO, P. **Modelagem Conceitual e Projetos de Bancos de Dados**. Rio de Janeiro: Editora Campus Ltda. ISBN 85-352-0158-0.

DEVAN, A. Impact Radius. **Impact Radius**, 2 abr. 2017. Disponivel em: <https://www.impactradius.com/blog/7-vs-big-data/>. Acesso em: 10 set. 2017.

ECMA-404. **The JavaScript Object Notation (JSON) Data Interchange Format**, ECMA Internacional**.** Disponivel em: < <https://www.rfc-editor.org/info/rfc7159>. Acesso em: 10 fev. 2017

FOWLER, M. **Aggregate Oriented Database**. MartinFowler, 19 jan. 2012. Disponivel em: <https://martinfowler.com/bliki/AggregateOrientedDatabase.html>. Acesso em: nov. 2017.

HEUSER, C. A. **Projeto de Bancos de Dados**. Porto Alegre: Sagra, 1998.

HOBBS, T. **Basic Rules of Cassandra Data Modeling**. Datastax, 2 fev. 2015. Disponivel em: <https://www.datastax.com/dev/blog/basic-rules-of-cassandra-data-modeling>. Acesso em: 15 jan. 2017.

HSIEH, D. **NoSQL Dat Modeling**. Ebay Inc, out. 2014. Disponivel em: <https://www.ebayinc.com/stories/blogs/tech/nosql-data-modeling/>. Acesso em: nov. 2017.

LEAVITT, N. **Will NoSQL Databases Live Up to Their Promise?** Computer, 2010.

LIMA, C. **Projeto Lógico de Bancos de Dados NOSQL Documentos a Partir de Esquemas Conceituais Entidade-Relacionamento Estandido(EER).** Universidade Federal de Santa Catarina, 2016.

MENNA, O. S.; RAMOS, L. A.; MELLO, R. S. **BrModeloNext: a Nova Versão de uma Ferramenta para Modelagem de Bancos de Dados Relacionais**. Sessão de Demos - XXVI SBBD, Florianópolis, 2011.

MIHALCEA, V. **How does MVCC (Multi-Version Concurrency Control) work**. Vladmihalcea, 1 mar. 2017. Disponivel em: <https://vladmihalcea.com/2017/03/01/how-does-mvcc-multi-version-concurrency-control-work/>.

MONGO. **Thinking in Documnets**. Mongodb, jan. 2015. Disponivel em: <https://www.mongodb.com/blog/post/thinking-documents-part-1>. Acesso em: 1 jun. 2017.

MONGO. **The MongoDB 3.4 Manual**. Mongodb, jun. 2017. Disponivel em: <https://docs.mongodb.com/manual/>. Acesso em: 1 nov. 2017.

MONGODB. **Top 5 Considerations When Evaluating NoSQL Databases**. MongoDB White Paper, nov. 2016.

MONIRUZZAMAN; HOSSAIN, S. **A. NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison**. International Journal of Database Theory and Application, Daffodil, v. 6, n. 4, p. 14, 2013.

MOZZILA. **JSON.** MDM Web Docs, abr. 2016. Disponivel em: <https://docs.mongodb.com/manual/>. Acesso em: 1 nov. 2018.

REUTER, N. **BrModeloNext.** GitHub. Disponivel em: < https://github.com/NathanReuter/brModeloNext>. Acesso em: 2 Jun. 2018.

NAVATHE&ELMASRI**. Fundamentals of Database-Systems 7th Edition**. 7th. ed. Arlington: Person, 2017.

NELSON, J. **Mastering Redis**. [S.l.]: [s.n.], 2016. ISBN-13: 978-1783988181.

O'Rourke, Brian P. **Lua: A Guide for Redis Users,** RedisGreen. Disponivel em: < https://www.redisgreen.net/blog/intro-to-lua-for-redis-programmers/>. Acesso em: 5 set. 2017.

ORGANIZATION, JSONSCHEMA. **Specifications.** JSONSchema, fev, 2016. Disponivel em: < http://json-schema.org/specification.html>. Acesso em: 2 fev. 2018.

PRITCHETT, D. **BASE: An Acid Alternative**. Magazine Queue - Object-Relational Mapping, 3 jun. 2008.

REDIS. **Redis Documentation**. Redis. Disponivel em: <https://redis.io/documentation>. Acesso em: 15 mar. 2017.

RUSSO, MICHEL. **Redis, from the Ground Up**, Mjrusso. Disponivel em: < http://blog.mjrusso.com/2010/10/17/redis-from-the-ground-up.html#heading_toc_j_45>. Acesso em: 20 abr. 2017.

SADALAGE, P. J. **NoSQL Distiled: A Brief Guide to the Emerging World of Polyglot Persistence**. Crowsfordsville: Addison-Wesley, 2012. ISSN 1098-. ISBN 9780321826626.

STRAUCH, C. **NoSQL Databases**. Estugarda: Hochschule der Medien, Stuttgart.

SPECIFICATION, BSON, **Bsonspec**. Bsonspec. Disponivel em: < http://bsonspec.org/spec.html>. Acesso em: 15 mar. 2018.

VERNON, V. **Modeling Aggregates with DDD and Entity Framework**, out. 2014. Disponivel em: < https://vaughnvernon.co/?p=879>. Acesso em: 15 nov. 2017.

# APÊNDICES

# APÊNDICE A – Regras e Códigos

1. **Modelo convertido para Mongodb**

```
use myDB

db.createCollection("author", {
  validator:{
    $jsonSchema: {
      bsonType: "object",
      properties:{
        _id: { bsonType: "objectId"},
        nome: { bsonType: "string"},
        email: { bsonType: "string"},
        enderecos: {
          bsonType: "array",
          minItems:1,
          items:[
          {
            bsonType: "object",
            properties:{
              rua: { bsonType: "string"},
            },
            additionalProperties : false,
          }
        ],
        },
        AtributoNoSQL: {
          bsonType: "array",
          minItems:1,
          items:{
            bsonType: "string"
          }
        },
        profile: {
          bsonType: "object",
          properties:{
            historico: { bsonType: "string"},
          },
          required: ["historico"],
          additionalProperties : false,
        },
        book_REF: {
          bsonType: "array",
          minItems:0,
          items:{
            bsonType: "objectId"
```

```
          }
        },
        dis1: {
          bsonType: "object",
          properties:{
            att: { bsonType: "string"},
          },
          required: ["att"],
          additionalProperties : false,
        },
        dis2: {
          bsonType: "object",
          properties:{
            att: { bsonType: "string"},
          },
          required: ["att"],
          additionalProperties : false,
        },
        phone: {
          bsonType: "array",
          minItems:0,
          items:{
            bsonType: "number"
          }
        },
        isStudent: { bsonType: "bool"},
      },
      required: ["_id", "email", "enderecos", "AtributoNoSQL",
"profile"],
      oneOf : [
        {required : ["dis1"]},
        {required : ["dis2"]},
      ],
    }
  },
  validationAction: "error",
  validationLevel: "moderate"
});

db.createCollection("book", {
  validator:{
    $jsonSchema: {
      bsonType: "object",
      properties:{
        name: { bsonType: "string"},
        _id: { bsonType: "objectId"},
        author_REF: { bsonType: "objectId"},
        ISBN: { bsonType: "string"},
        description: { bsonType: "string"},
```

```
        Publisher_REF: { bsonType: "objectId"},
      },
      required: ["name", "_id", "author_REF", "ISBN",
"Publisher_REF"],
    }
  },
  validationAction: "error",
  validationLevel: "moderate"
});

db.createCollection("Publisher", {
  validator:{
    $jsonSchema: {
      bsonType: "object",
      properties:{
        _id: { bsonType: "objectId"},
        name: { bsonType: "string"},
        location: { bsonType: "string"},
        book_REF: {
          bsonType: "array",
          minItems:0,
          items:{
            bsonType: "objectId"
          }
        },
      },
      required: ["_id", "name", "location"],
    }
  },
  validationAction: "error",
  validationLevel: "moderate"
});
```

### 2. Modelo Convertido de Coleção Única para MongoDB

```
use NovoDB
db.createCollection("SingleCollection", {
  validator:{
    $jsonSchema: {
      bsonType: "object",
      properties:{
        author: {
          bsonType: "object",
          properties:{
            _id: { bsonType: "objectId"},
            nome: { bsonType: "string"},
            email: { bsonType: "string"},
            enderecos: {
```

```
  bsonType: "array",
  minItems:1,
  items:[
  {
    bsonType: "object",
    properties:{
      rua: { bsonType: "string"},
    },
    additionalProperties : false,
  }
],
},
AtributoNoSQL: {
  bsonType: "array",
  minItems:1,
  items:{
    bsonType: "string"
  }
},
profile: {
  bsonType: "object",
  properties:{
    historico: { bsonType: "string"},
  },
  required: ["historico"],
  additionalProperties : false,
},
book_REF: {
  bsonType: "array",
  minItems:0,
  items:{
    bsonType: "objectId"
  }
},
dis1: {
  bsonType: "object",
  properties:{
    att: { bsonType: "string"},
  },
  required: ["att"],
  additionalProperties : false,
},
dis2: {
  bsonType: "object",
  properties:{
    att: { bsonType: "string"},
  },
  required: ["att"],
  additionalProperties : false,
```

```
            },
            phone: {
              bsonType: "array",
              minItems:0,
              items:{
                bsonType: "number"
              }
            },
            isStudent: { bsonType: "bool"},
          },
          required: ["_id", "email", "enderecos", "AtributoNoSQL",
"profile"],
          additionalProperties : false,
        },
        book: {
          bsonType: "object",
          properties:{
            name: { bsonType: "string"},
            _id: { bsonType: "objectId"},
            author_REF: { bsonType: "objectId"},
            ISBN: { bsonType: "string"},
            description: { bsonType: "string"},
            Publisher_REF: { bsonType: "objectId"},
          },
          required: ["name", "_id", "author_REF", "ISBN",
"Publisher_REF"],
          additionalProperties : false,
        },
        Publisher: {
          bsonType: "object",
          properties:{
            _id: { bsonType: "objectId"},
            name: { bsonType: "string"},
            location: { bsonType: "string"},
            book_REF: {
              bsonType: "array",
              minItems:0,
              items:{
                bsonType: "objectId"
              }
            },
          },
          required: ["_id", "name", "location"],
          additionalProperties : false,
        },
      },
      required: ["book", "Publisher"],
    }
  },
```

```
  validationAction: "error",
  validationLevel: "moderate"
});
```

### 3. Código para o caso correto de inserção na coleção única do mongo

```
db.SingleCollection.insert({
    "Publisher": {
        "_id": ObjectId("5b072487311587414504 71b8"),
        "name": "UFSC",
        "location": "Santa Catarina"
    },
    "author": {
        "_id": ObjectId("5b0724d631158741450471b9"),
        "nome": "Nathan",
        "email": "nathan@email.com",
        "enderecos": [{
            "rua": "Barauna"
        }],
        "AtributoNoSQL": [
            "atributo1"
        ],
        "profile": {
            "historico": "Histórico"
        },
        "dis1": {
            "att": "Atributo"
        }
    },
    "book": {
        "_id": ObjectId("5b0726a331158741450471ba"),
        "name": "Guia do Mochileiro das Galaxias",
        "ISBN": "9781234567897",
        "Publisher_REF": ObjectId("5b072487311587414504 71b8"),
        "author_REF": ObjectId("5b0724d631158741450471b9")
    }
})
```

### 4. Código para o caso incorreto de inserção na coleção única do mongo

```
db.SingleCollection.insert({
    "Publisher": {
        "_id": ObjectId("5b072487311587414504 71b8"),
        "name": "UFSC",
        "location": "Santa Catarina"
    },
    "author": {
        "_id": ObjectId("5b0724d631158741450471b9"),
```

```
        "nome": "Nathan",
        "enderecos": [{
            "rua": 1
        }],
        "AtributoNoSQL": [
            "atributo1"
        ],
        "profile": {
            "historico": "Histórico"
        },
        "dis1": {
            "att": "Atributo"
        }
    },
    "book": {
        "name": "Guia do Mochileiro das Galaxias",
        "ISBN": "9781234567897",
        "Publisher_REF": ObjectId("5b0724873115874145 0471b8"),
        "author_REF": ObjectId("5b0724d631158741450471b9")
    }
})
```

### 5. Modelo convertido para Cassandra

```
CREATE KEYSPACE IF NOT EXISTS NovoDB WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor':'3'
};
use NovoDB;

CREATE TYPE ENDERECOS (
  rua TEXT,

);
CREATE TYPE PROFILE (
  historico TEXT,

);
CREATE TYPE DIS1 (
  att TEXT,

);
CREATE TYPE DIS2 (
  att TEXT,

);
CREATE TABLE author (
```

```
  id UUID,
  nome TEXT,
  email TEXT,
  enderecos list<frozen <ENDERECOS>>,
  AtributoNoSQL list<TEXT>,
  profile PROFILE,
  book_REF list<UUID>,
  dis1 DIS1,
  dis2 DIS2,
  phone list<INT>,
  isStudent BOOLEAN,
  PRIMARY KEY (id)
);
CREATE TABLE book (
  name TEXT,
  id UUID,
  author_REF UUID,
  ISBN TEXT,
  description TEXT,
  Publisher_REF UUID,
  PRIMARY KEY (id)
);
CREATE TABLE Publisher (
  id UUID,
  name TEXT,
  location TEXT,
  book_REF list<UUID>,
  PRIMARY KEY (id)
);
```

### 6. Código de Inserção no Cassandra

```
INSERT INTO publisher (id , book_ref , location , name )
     VALUES ( 34703a2e-e844-4be3-9be1-341d95a7f019 , [efd82c1f-eb25-
4c87-a586-92acf680739b], 'SC','UFSC');

INSERT INTO author (id, nome , email, enderecos, atributonosql ,
profile , book_ref,  dis1, phone , isstudent)
VALUES ( e95a78a2-4b1f-46d8-8a77-59146e45e3d7, 'Nathan',
'nathan@mail.com',[{rua: 'Rua Barauna'}], ['teste'], {historico:
'Algum histórico'}, [efd82c1f-eb25-4c87-a586-92acf680739b], {att:
'dis1'}, [999887755, 884477332], TRUE);

INSERT INTO book (id , name , author_ref , isbn , description ,
publisher_ref ) VALUES ( efd82c1f-eb25-4c87-a586-92acf680739b, 'Guia
do Mochileiro das Galaxias', e95a78a2-4b1f-46d8-8a77-59146e45e3d7,
'9781234567897', 'Good book', 34703a2e-e844-4be3-9be1-341d95a7f019);
```

## 7. Código em lua para teste no Redis

```
local function split(inputstr, sep)
        if sep == nil then
                sep = "%s"
        end
        local t={} ; local i=1
        for str in string.gmatch(inputstr, "([^"..sep.."]+)") do
                t[i] = str
                i = i + 1
        end
        return t
end

local function findCollection(colToInsert)
  local colName = split(colToInsert, ":")[1]
  return colName
end

local function CheckRequiredAttributes(colection_required, length)
  local isCorrect = true

  if (table.getn(KEYS) - 1  < length) then
    return false
  end

  for _,key in ipairs(KEYS) do
    if (_ > 1) then
      if (not colection_required[key]) then
        isCorrect = false
      end
    end
  end

  return isCorrect
end

local function prettyPrinter(result)
  if (result) then
    return 'Schema is Correct'
  end

  return 'Error in validation'
end

local function checkCollectionCase (colName)
  if (colName == 'author') then
```

```
    local required = {id = true,  email = true,  enderecos = true,
AtributoNoSQL = true,  profile = true,  dis1 = true,  dis2 = true,
}
    return CheckRequiredAttributes(required, 7)
  elseif (colName == 'book') then
    local required = {name = true,  id = true,  author_REF = true,
ISBN = true,  Publisher_REF = true,  }
    return CheckRequiredAttributes(required, 5)
  elseif (colName == 'Publisher') then
    local required = {id = true,  name = true,  location = true,  }
    return CheckRequiredAttributes(required, 3)
  end
end

return prettyPrinter(checkCollectionCase(findCollection(KEYS[1])));
```

### 8. Código Teste para Sucesso Redis

```
redis-cli --eval redisTestScript.lua book id name author_REF ISBN
Publisher_REF
```

### 9. Código Teste para Erro Redis

```
redis-cli --eval redisTestScript.lua book id name author_REF ISBN
```