Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Banco de Dados Cassandra: Um Estudo de Caso para Análise dos Dados dos Servidores Públicos Federais

Lizane Alvares Leite

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.ª Dr.ª Maristela Terto de Holanda

Coorientador
Ms. Ruben Cruz Huacarpuma

Brasília
2014

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Homero Luiz Piccolo

Banca examinadora composta por:

Prof.ª Dr.ª Maristela Terto de Holanda (Orientadora) — CIC/UnB
Prof.ª Dr.ª Genaina Nunes Rodrigues — CIC/UnB
Prof. Ms. Henrique Pereira de Freitas Filho — IFB

CIP — Catalogação Internacional na Publicação

Leite, Lizane Alvares.

Banco de Dados Cassandra: Um Estudo de Caso para Análise dos Dados dos Servidores Públicos Federais / Lizane Alvares Leite. Brasília : UnB, 2014.
99 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2014.

1. banco de dados, 2. nosql, 3. cassandra, 4. servidor publico

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Banco de Dados Cassandra: Um Estudo de Caso para Análise dos Dados dos Servidores Públicos Federais

Lizane Alvares Leite

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.ª Dr.ª Maristela Terto de Holanda (Orientadora)
CIC/UnB

Prof.ª Dr.ª Genaina Nunes Rodrigues
CIC/UnB

Prof. Ms. Henrique Pereira de Freitas Filho
IFB

Prof. Dr. Homero Luiz Piccolo
Coordenador do Bacharelado em Ciência da Computação

Brasília, 4 de Setembro de 2014

Dedicatória

Ao meus pais, que sempre me incentivam à atingir os meus objetivos por meio da busca incessante por conhecimento.

Agradecimentos

Agradeço à Deus por tudo, nada seria possível sem Ele. Agradeço, também, aos meus familiares, amigos, namorado e professores pela paciência e por acreditarem na realização deste trabalho.

Resumo

Um banco de dados NoSQL é um banco não relacional, distribuído e rápido, que trabalha com grande volume de dados. Em geral, os bancos de dados NoSQL são uma alternativa para bancos de dados relacionais, com características como escalabilidade, disponibilidade e tolerância a falhas. Utilizam um modelo de dados muito flexível, livre de esquemas, com escalabilidade horizontal e arquitetura distribuída. Este trabalho apresenta um estudo de caso para analisar dados dos servidores públicos federais, utilizando o Cassandra, um banco de dados NoSQL. Este estudo expõe testes feitos para avaliar a eficiência da inserção e recuperação de dados entre alguns modelos para o Cassandra, comparando-os com o modelo de dados real implementado em PostgreSQL.

Palavras-chave: banco de dados, nosql, cassandra, servidor publico

Abstract

A NoSQL database is not a relational database, distributed and fast, working with large amount of data. In general, NoSQL databases are an alternative to relational databases, with features like scalability, availability and fault tolerance. Use a very flexible data model, scheme-less, with horizontal scalability and distributed architecture. This paper presents a case study to analyze data from federal public employees, Cassandra using a NoSQL database. This study exposes tests to evaluate the efficiency of insertion and retrieval of data between models for Cassandra, comparing them with the real data model implemented in PostgreSQL.

Keywords: database, nosql, cassandra, public employee

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O Sistema Integrado de Administração de Recursos Humanos (SIAPE) utiliza um banco de dados objeto-relacional para fazer a integração de todas as plataformas de gestão de folha de pagamento dos servidores públicos, sendo considerado um dos principais sistemas que estruturam o governo do Brasil [6]. O pagamento dos servidores públicos é composto por várias rubricas, onde cada uma dessas rubricas representa um tipo de pagamento. Por exemplo, um servidor que recebe apenas o vencimento básico e adicional noturno, recebe então duas rubricas, uma para cada tipo de pagamento.

Além da gestão de pagamentos, os dados do SIAPE também são importantes para fazer trilhas de auditoria que verificam a consistência das informações deste banco. Uma destas trilhas é chamada de trilha de auditoria para incompatibilidade de rubrica. Tal trilha verifica se existe algum servidor que recebe rubricas incompatíveis, ou seja, rubricas que não podem ser recebidas simultaneamente. A rotina para verificar a incompatibilidade de rubricas é executada no conjunto de dados referente a um mês de pagamento e normalmente demora muito tempo.

Neste trabalho é proposta uma nova maneira de implementação do banco de dados com um paradigma diferente do relacional com o objetivo de melhorar o tempo das consultas para verificação de compatibilidade de rubrica. Para tanto, será utilizado um banco de dados NoSQL para armazenar os dados dos servidores e fazer as consultas para esta trilha de auditoria.

Um banco de dados NoSQL é um banco não relacional, distribuído e rápido, que trabalha com grande volume de dados. Em geral, os bancos de dados NoSQL são uma alternativa para bancos de dados relacionais, com características como escalabilidade, disponibilidade e tolerância a falhas. Utilizam um modelo de dados muito flexível, livre de esquemas, com escalabilidade horizontal e arquitetura distribuída. Do ponto de vista do negócio, considerar o uso de um banco de dados NoSQL tem trazido uma grande vantagem competitiva em vários ramos da indústria. Existem vários casos de sucesso que utilizaram o conceito NoSQL, dentre eles: Google, Facebook e Amazon [9].

Existem vários sistemas gerenciadores de bancos de dados NoSQL, sendo que neste trabalho é utilizado o Cassandra. O Cassandra é um projeto do Apache que nasceu no Facebook e sua construção foi baseada no Dynamo da Amazon e no BigTable do Google (ambos são sistemas gerenciadores de bancos de dados NoSQL). Cassandra é um sistema de armazenamento distribuído para o gerenciamento de grandes quantidades de dados,

utilizando para tanto vários servidores, oferecendo um serviço altamente disponível sem nenhum ponto de falha [8].

Neste contexto, o presente trabalho apresenta um estudo sobre os dados dos servidores públicos federais para verificação de incompatibilidade de rubricas utilizando o Cassandra. O Cassandra tem sido utilizado em estudos tanto no meio acadêmico, quanto na indústria e tem mostrado resultados satisfatórios.

1.1 Problema e Hipótese

O problema foco deste trabalho é o tempo gasto na consulta da trilha de auditoria para investigar a incompatibilidade de rubricas, ou seja, verificar pagamentos inapropriados para servidores públicos provenientes do recebimento simultâneo de rubricas que não são compatíveis. A hipótese é que com uma modelagem de dados para o Cassandra, seja possível melhorar o tempo de processamento da consulta desta trilha de auditoria.

1.2 Motivação

O uso de bancos de dados NoSQL é recente, mas tem mostrado bons resultados para aplicações compatíveis com este modelo. Empresas como Google têm feito estudos sobre este paradigma e obtido bons resultados em seus negócios. Porém, não existem muitos estudos que envolvam o uso do NoSQL no contexto da administração publica. A motivação deste trabalho, portanto, se dá na oportunidade de estudar um caso real, trazendo soluções diferentes do que é utilizado atualmente neste ramo, além de fazer a análise prática de um banco de dados NoSQL relativamente novo.

1.3 Objetivo

1.3.1 Objetivo Geral

O objetivo geral deste trabalho é analisar o comportamento do banco de dados Cassandra com os dados do SIAPE, desenvolvendo um modelo de dados adequado para a busca de incompatibilidade de rubricas e verificando a performance desta consulta.

1.3.2 Objetivo Especifico

Para alcançar o objetivo geral deste trabalho, os seguintes objetivos específicos foram necessários:

- Definir uma metodologia para criar e manter um ambiente Cassandra, incluindo a escolha do sistema operacional, linguagem de programação, ferramentas, utilitários e modelos de dados a serem utilizados.
- Fazer um estudo de caso com os dados do SIAPE, testando a eficiência da inserção de dados e da busca de dados no ambiente Cassandra e fazendo um comparativo com o desempenho da implementação atual com a implementação proposta.

- Fazer testes utilizando possíveis modelos de dados do Cassandra e fazendo comparações entre eles.

1.4 Estrutura do Trabalho

Este trabalho é estruturado da seguinte forma:

- Capítulo 2: É feita uma introdução aos bancos de dados, explicando o que é um sistema gerenciador de banco de dados e conceitos importantes, além de abordar a definição de banco de dados relacional, elucidando características relevantes do modelo de dados relacional, normalização de dados e propriedades ACID. Bancos de dados não relacionais também são apresentados, discutindo-se o modelo chave-valor, o modelo orientado a coluna, o modelo orientado a documento, e explanando os pilares do Teorema CAP.
- Capítulo 3: O banco de dados Cassandra é apresentado enfatizando suas principais característica e é abordado o modelo de dados com os conceitos de *clustes*, *keyspace*, família de colunas e coluna. A arquitetura do Cassandra também é exposta, explicando como é feita a comunicação entre nós, como os dados são distribuídos e replicados no ambiente, quais são os particionadores disponíveis para o uso, o que são *snitches* e como funciona solicitações de clientes.
- Capítulo 4: O Sistema Integrado de Administração de Recursos Humanos é apresentado, explicando em qual contexto ele se insere na administração publica e expondo o modelo de dados utilizado no ambiente PostgreSQL. O modelo proposto é abordado, bem como as etapas necessárias para implementação e configuração do ambiente Cassandra e aplicações que foram necessárias.
- Capítulo 5: Os resultados obtidos nos testes de carga de dados utilizando os clientes Cassandra BulkLoader e o Cassandra JDBC são exibidos, além dos testes da consulta da trilha de auditoria para incompatibilidade de rubricas realizados com cliente Hector com uma única *thread* e com várias *threads*. E também, é apresentada uma comparação entre o modelo de dados proposto e o real.
- Capítulo 6: Por fim, é feito um fechamento recapitulando pontos importantes discutidos ao longo do trabalho. Bem como, a conclusão obtida e possíveis trabalhos futuros.

Capítulo 2

Banco de Dados

Neste capítulo, é feita uma abordagem geral sobre bancos de dados relacionais e não relacionais. Na Seção 2.1, define-se o conceito de banco de dados e de sistema gerenciador de banco de dados. Posteriormente na Seção 2.2, é feita uma introdução ao modelo de dados relacional, onde se aborda conceitos básicos, e há uma breve explicação sobre normalização e propriedades ACID. Na Seção 2.3, é feita uma introdução aos modelos não relacionais, discutindo-se o modelo chave-valor, modelo orientado a coluna, modelo orientado a documento, além de uma rápida abordagem ao Teorema CAP.

2.1 Sistema Gerenciador de Banco de Dados

Banco de dados é um conjunto de dados integrados que atendem a um conjunto de sistemas [13], que possuem um significado implícito e podem ser gravados. De forma menos genérica, Elmasri e Navathe [11] definem que um banco de dados reproduz alguma característica do mundo real, podendo ser chamado de minimundo ou universo de discurso. Além disso, um banco de dados é um conjunto lógico e coerente de dados com um significado inerente sendo projetado, construído e populado por dados que atendem a uma propósito específico. Um banco de dados possui um grupo de usuários definido e algumas aplicações delineadas com os interesses desse grupo de usuários [11].

Um banco de dados poderia ser criado e mantido manualmente por registros de papel ou por arquivos de texto, porém para sistemas computacionais atuais geralmente é utilizado um Sistema Gerenciador de Banco de Dados (SGBD). Um sistema de software de propósito geral que auxilia o usuário a definir, construir, manipular e compartilhar bancos de dados entre usuários e sistemas é chamado de SGBD. Neste contexto, definir um banco de dados é especificar os tipos de dados, estruturas e restrições para os dados que serão guardados. Já construir um banco de dados é o processo de armazenar os dados em algum lugar apropriado que será controlado pelo SGBD. Sendo que manipular um banco de dados é recuperar um determinado dado, atualizar o banco de dados para reproduzir mudanças no minimundo e gerar relatórios dos dados. E finalizando, compartilhar um banco de dados é permitir aos usuários e aplicações acessar o banco de dados simultaneamente [11].

Existem muitas funcionalidades que um SGBD deve possuir, algumas delas são [11]:

- Controle de redundância;

- Restrição de acesso não autorizado;
- Garantia de armazenamento persistente para objetos programas;
- Garantia de armazenamento de estruturas para o processamento eficiente de consultas;
- Garantia de *backup* e restauração;
- Fornecimento de múltiplas interfaces para os usuários;
- Representar relacionamentos complexos entre os dados;
- Forçar as restrições de integridade;
- Permitir interferências e ações usando regras.

Existem algumas situações em que o uso de um SGBD não é indicado, como por exemplo quando envolve custos altos e desnecessários, sendo mais indicado o uso tradicional de arquivos. Dentre as situações que aumentam o custo da utilização de um SGBD, estão os investimentos iniciais em hardware, software e treinamento, além dos custos elevados para oferecer segurança, controle de concorrência, recuperação e restrição de integridade. Também pode ser indicado o uso de arquivos convencionais quando o banco de dados e suas aplicações são muito simples, bem definidas, sem previsão de mudanças e quando não é necessário o acesso de múltiplos usuários aos dados [11].

2.2 Modelo Relacional

O modelo de dados relacional foi apresentado por Codd em 1970, época onde os sistemas de banco de dados eram baseados ou no modelo hierárquico ou no modelo de rede. O modelo relacional inovou a área de banco de dados, sobrepujando os outros. Vários SGBDs foram prototipados na década de 70 por projetos de pesquisa da IBM e da Universidade da California em Berkeley [16]. O modelo relacional é muito utilizado atualmente e compatível com aplicações que tem dados estruturados, inter-relacionados e referenciados de forma bem definida [18].

2.2.1 Definição

Um banco de dados relacional é formado por um conjunto de relações. Por sua vez, uma relação é um conjunto não ordenado de tuplas e uma tupla é um conjunto de valores de atributos. Seguindo, um valor de atribulo é identificado pelo nome de atributo e um atributo é definido como um conjunto de valores de atributos das tuplas de uma relação, que possuem o mesmo nome de atributo [13].

A Figura 2.1 ilustra o exemplo de uma relação chamada Emp. Ela possui 4 atributos, cujos os nomes de atributos são: CodigoEmp, Nome, CodigoDepto e CategFuncional. A tupla marcada no desenho possui o seguinte conjunto de valores de atributos: E3, Santos, D2 e C5. O atributo CodigoDepto possui o seguinte conjunto de valores de atributos: D1, D2, D1 e D1.

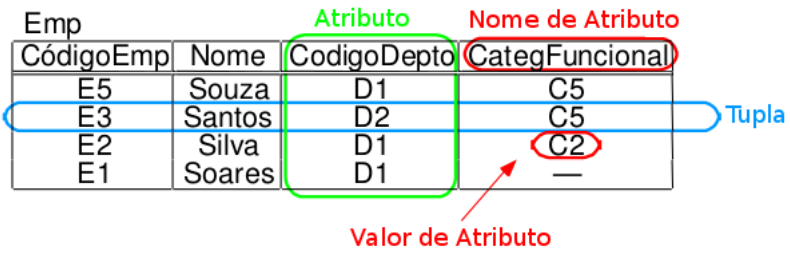

Figura 2.1: Exemplo de uma relação [13]

2.2.2 Chaves e Domínio

O relacionamento entre tuplas de relações é formado pela chave. Existem três tipos de chaves [13]: chave primária, chave estrangeira e chave alternativa.

Chave primária: É um atributo ou combinação de atributos que possuem valores de atributo que identificam unicamente uma tupla de uma relação. Uma chave primária deve ser mínima, ou seja, os seus atributos devem ser os mínimos necessários que garantem a unicidade do valor da chave [13].

Chave estrangeira: É um atributo ou combinação de atributos que necessariamente possuem valores de atributo de uma chave primária de uma relação. A chave estrangeira permite implementar relacionamentos entre relações por referenciar um valor de atributo existente. É importante salientar que a referência não é restringida à outra relação, ou seja, a chave estrangeira pode fazer menção à uma chave primária de sua própria relação [13].

Chave alternativa: É um atributo ou combinação de atributos que possuem valores de atributo que identificam unicamente uma tupla de uma relação, mas que não foi eleita como chave primária [13].

Quando uma relação é definida, deve ser especificado um conjunto de valores que cada atributo da relação pode assumir. Este conjunto de valores é chamado de domínio do atributo. Deve ser especificado, também, se o atributo pode ou não ser vazio, significando que o atributo pode ou não receber nenhum valor do seu domínio. São chamadas de atributos obrigatórios aqueles que não admitem valor vazio. E são chamados de atributos opcionais aqueles que admitem valor vazio. Em geral, chaves primárias e chaves estrangeiras são atributos obrigatórios [13].

2.2.3 Restrições

Normalmente em um banco de dados relacional, existirão várias tuplas relacionadas com outras tuplas de diversas maneiras. Sendo que o estado do banco de dados corresponderá aos estados de todas as suas relações em determinado instante. Por isso, existem limitações e restrições para os valores reais em um estado do banco de dados derivados de

regras do minimundo que o bando de dados representa [11]. Algumas restrições que são impostas pela aplicação estão descritas a seguir [13] [11]:

- Restrição de integridade de domínio garante que o valor de atributo obedeça a definição do domínio do atributo.
- Restrição de integridade de vazio garante que o atributo obedeça a especificação de admitir ou não valor vazio.
- Restrição de integridade de chave garante que o valor de atributo de chave primária seja único e não vazio.
- Restrição integridade referencial garante que o valor de atributo de chave estrangeira seja igual ao valor atributo de uma chave primária.

Existem também restrições inerentes ao modelo relacional, e outras que são expressas no esquema do modelo relacional por meio da definição dos dados [11].

2.2.4 Forma Normal

O processo de normalização foi proposto inicialmente por Codd em 1972. Este é um processo para certificar que um conjunto de relações satisfaçam a forma normal, avaliando as relações sob os critérios de cada forma normal. Codd propôs três formas normais intituladas de primeira, segunda e terceira forma normal. A normalização de dados pode ser vista como um processo de análise para se alcançar a minimização de redundância e minimização de anomalias de inserção, exclusão e atualização [11].

Primeira Forma Normal: A primeira forma normal estabelece que o domínio de um atributo deve incluir somente valores atômicos e que o valor de qualquer atributo em uma tupla deve ter um único valor no domínio daquele atributo. Isto é, não pode existir como valor de atributo de uma única tupla um conjunto de valores, uma tupla de valores ou uma combinação entre ambos [11].

Segunda Forma Normal: A segunda forma normal estipula que todo atributo que não é chave primária deve ter dependência funcional total da chave primária da relação, além de satisfazer a primeira forma normal. Ou seja, um atributo que não é chave primária não pode depender de parte da chave primária [11].

Terceira Forma Normal: A terceira forma normal dispõe que nenhum atributo que não é chave primária deve ser transitivamente dependente da chave primária, além de satisfazer a segunda forma normal [11]. Ou seja, um atributo que não é chave primária não pode depender de outro atributo ou combinação de atributos que não são chaves primárias [13].

2.2.5 Propriedades ACID

Uma transação é um programa em execução que forma uma unidade lógica de processamento no banco de dados, incluindo neste uma ou mais operações de acesso, englobando

operações de inserção, alteração ou recuperação. As operações de banco de dados que formam uma transação podem estar embutidas em um programa de aplicação ou podem ser especificadas interativamente, via linguagem de consulta de alto nível [11].

Se uma transação for executada por um SGBD, o sistema deverá garantir que todas as operações foram completadas com sucesso e que seu efeito será gravado permanentemente no banco de dados ou que a transação não terá nenhum efeito sobre o banco de dados ou sobre outras transações. Ou seja, o SGBD não deverá permitir que algumas das operações de uma transação sejam aplicadas e outras não [11].

O SGBD, também, não deverá permitir que transações concorrentes sejam executadas de maneira descontrolada. Pode ocorrer, entretanto, o problema de atualização perdida, que acontece quando duas transações acessam o mesmo ítens de banco de dados e têm suas operações intercaladas, tornando o valor de alguns dos ítens do banco de dados incorretos. Ou o problema de atualização temporária, que acontece quando uma transação atualiza um item de banco de dados, falha por algum motivo e antes que o item retorne ao valor original outra transação acessa o item atualizado [11].

As transações devem possuir algumas particularidades que são chamadas propriedades ACID. Elas são impostas pelo controle de concorrência e métodos de restauração do SGBD [11] e são definidas, conforme Elmasri e Navathe, da seguinte maneira:

- Atomicidade: Uma transação é uma unidade atômica de processamento, sendo ela é executada em sua totalidade ou não é executada de forma alguma.
- Preservação de consistência: Uma transação preservará a consistência, se sua execução completa fizer o banco de dados passar de um estado consistente para outro estado consistente.
- Isolamento: Uma transação deverá ser executada isoladamente das demais transações, ou seja, a execução de uma transação não deve sofrer interferência de nenhuma outra transação concorrente.
- Durabilidade ou permanência: As mudanças aplicadas ao banco de dados por uma transação efetivada deverão persistir no banco de dados, isto é, mudanças não devem ser perdidas em razão de uma falha.

As propriedades ACID são implementadas em Sistemas Gerenciadores de Bancos de Dados Relacionais (SGBDRs). Muitas vezes, os *frameworks* e linguagens de programação que trabalham com SGBDRs tentam estender as propriedades ACID para toda a aplicação, e operam satisfatoriamente para bancos de dados em ambientes com apenas um único servidor ou nó. Porém, se torna complexo manter as propriedades ACID quando é utilizado vários nós [18].

2.3 Modelo Não Relacional

NoSQL significa *Not Only SQL*, ou seja, não somente *SQL*. NoSQL é usado como um termo genérico para todos os bancos de dados e armazenamento de dados que não seguem os princípios bem estabelecidos dos SGBDRs e muitas vezes estão relacionados com grandes conjuntos de dados acessados e manipulados na *web*. Isso significa que

NoSQL representa uma classe de produtos e uma coleção de diversos conceitos, muitas vezes relacionado ao armazenamento de dados e sua manipulação [18].

Os desafios de SGBDRs para o processamento de dados massivos não são específicos de um produto, mas dizem respeito a toda a classe de bancos de dados relacionais. SGBDR assume uma estrutura bem definida de dados onde esses dados são massivos e largamente uniformes. SGBDR baseia-se no pré-requisito de que as propriedades dos dados podem ser definidas antecipadamente e que as suas inter-relações estão bem estabelecidas e sistematicamente referenciadas. Ele também assume que os índices podem ser consistentemente definido em conjuntos de dados e que tais índices podem ser uniformemente utilizados para uma rápida consulta. Porém, o SGBDR costuma ser pouco útil quando estes pressupostos não são verdadeiros. SGBDR poderia lidar com algumas irregularidades e falta de estrutura, como a desnormalização de tabelas, exclusão de restrições e diminuição de garantia transacional, mas com essas modificações o SGBDR perde suas principais características. O NoSQL ameniza estes problemas que o SGBDR impõe e torna mais fácil trabalhar com grandes quantidades de dados esparsos, mas por sua vez, tiram o poder da integridade transacional, flexibilidade de indexação e consultas [18].

As principais vantagens dos bancos de dados NoSQL são [12]:

- Leitura e escrita de dados rápida;
- Apoio ao armazenamento de dados massivos;
- Fácil expansão de dados;
- Baixo custo.

Em contrapartida, as deficiências do NoSQL são [12] [18]:

- Falta de suporte à linguagem de consulta estruturada;
- Carência no gerenciamento transacional.

Os bancos de dados tradicionais normalmente utilizam o modelo de dados relacional, principalmente pelo suporte às propriedades transacionais ACID. Mas, no âmbito de bancos de dados NoSQL, os modelos de dados predominantes são: modelo chave-valor, modelo orientado a coluna e modelo orientado a documento [12].

2.3.1 Modelo Chave-valor

O modelo de armazenamento chave-valor é um modelo muito simples. Este modelo pareia chaves com valores da mesma forma que um mapa ou tabela *hash* faz nas linguagens de programação populares. Algumas implementações do modelo chave-valor permitem tipos de valores complexos como listas ou fornecem um meio de iteração através das chaves, não sendo características intrínsecas do modelo. Como o modelo de armazenamento chave-valor não é muito exigente, os bancos de dados deste tipo podem ter um alto desempenho em vários cenários, mas geralmente não são eficazes quando existe a necessidade fazer consultas e agregações mais complicadas. Chave-valor mapeia chaves simples para valores possivelmente mais complexos, como uma grande tabela *hash*. Devido à sua simplicidade, este gênero de banco de dados tem uma boa flexibilidade de implementação. Porém, sua simplicidade pode ser uma desvantagem para dados com requisitos de modelagem complexas [17].

Vantagem: O armazenamento chave-valor é projetado para ter escalabilidade horizontal e rapidez, mas em contextos onde a necessidade de manter índices é inexistente ou pequena. Particularmente, é adequado para problemas em que os dados não são altamente relacionados. Por exemplo, uma aplicação *web* que atende esses critérios é o armazenamento de dados de sessão dos usuários, já que as atividades de sessão de um usuário serão diferentes e normalmente não relacionadas com as atividades de outros usuários [17].

Desvantagem: O armazenamento chave-valor não é útil se for necessário realizar consultas complexas sobre os dados, pois o modelo não tem muita habilidade com índices. Faz somente operações básicas de leitura e escrita [17].

Exemplos: Alguns SGBDs que implementam o modelo de armazenamento chave-valor são: Voldemort, Redis e Riak [17].

2.3.2 Modelo Orientado a Coluna

O modelo de armazenamento orientado a coluna é chamado assim, pois os dados de uma determinada coluna (tabela bidimensional) são armazenados juntos. Por outro lado, um banco de dados orientado a linha, assim como um banco de dados relacional, mantém as informações sobre uma tupla juntas. A diferença pode parecer irrelevante, mas o impacto desta decisão de projeto é importante. Em bancos de dados orientados a coluna, a adição de colunas é muito barata e é feita linha por linha. Cada linha pode ter um conjunto diferente de colunas, inclusive nenhuma, permitindo que as tabelas permaneçam vazias, sem implicar em custos de armazenamento para valores nulos [17].

Vantagem: O modelo de armazenamento orientado a coluna têm sido tradicionalmente desenvolvido para ter escalabilidade horizontal. Assim, eles são particularmente adequados para problemas de *Big Data* utilizando grupos de vários nós. Eles também tendem a ter suporte para recursos como compressão e controle de versão [17].

Desvantagem: O projeto do esquema do banco de dados orientado a coluna deve ser planejado de acordo com as consultas dos dados. Isto significa que antes de projetar o banco, deve-se saber a forma como os dados serão utilizados e não apenas como vão consistir. Se os padrões de uso dos dados não pode ser definido antes do projeto do banco, o modelo orientado a coluna não será uma boa escolha [17].

Exemplos: Alguns SGBDs que implementam o modelo de armazenamento orientando a coluna são: Cassandra, HBase e Hypertable [17].

2.3.3 Modelo Orientado a Documento

O modelo de armazenamento orientado a documentos é como um *hash* com um campo identificador único e valores que podem ser de vários tipos. Os documentos podem conter estruturas aninhadas, apresentando flexibilidade e permitindo domínios variáveis. O modelo impõe poucas restrições sobre os dados de entrada, sendo necessário, basicamente, que seja um documento. Normalmente, cada SGBD orientado a documentos tem uma

abordagem diferente em relação a indexação, a consultas *ad hoc*, a replicação, a consistência e a outras decisões de projeto. Escolher entre eles requer a compreensão dessas diferenças e como elas impactam no casos de uso específicos [17].

Vantagem: Bancos de dados orientados a documento são adequados para os problemas que envolvem domínios altamente variáveis. Ele é uma boa escolha quando não se sabe exatamente como são os dados. Além disso, devido à natureza dos documentos, este modelo é normalmente bem compatível com a programação orientada a objetos. Ou seja, há menos diferenças na abstração dos dados entre o modelo de banco de dados e da aplicação [17].

Desvantagem: Um documento geralmente deve conter a maioria ou todas as informações relevantes necessárias para o seu uso. Assim, é normal a utilização de dados desnormalizados em um banco de dados orientado a documentos; enquanto que em um banco de dados relacional, a normalização é crucial para reduzir ou eliminar as redundância e cópias que podem ficar fora de sincronização [17].

Exemplos: Alguns SGBDs que implementam o modelo de armazenamento orientado a documento são: MongoDB e CouchDB [17].

2.3.4 Teorema CAP

Consistência, disponibilidade e tolerância ao particionamento são os três pilares do teorema CAP que estão subjacentes a grande parte da recente geração de pensamentos em torno de integridade transacional em grandes sistemas distribuídos e escaláveis. Sucintamente, o teorema CAP afirma que em sistemas que são distribuídos ou escaláveis, é impossível atingir todos os três pilares ao mesmo tempo. Deve ser feita uma escolha e renunciar a pelo menos um pilar em favor dos outros dois [18]. Ou seja, é possível ter um banco de dados distribuído e escalável, que é consistente e tem tolerância ao particionamento, mas não é disponível. Ou um sistema que é disponível e tem tolerância ao particionamento, mas não é consistente. Ou um sistema que é consistente e disponível, mas não tem tolerância ao particionamento [17].

Consistência: No contexto do teorema CAP, consistência remete à atomicidade e isolamento. Consistência significa ler e escrever consistentemente, de modo que operações concorrentes vejam o mesmo estado de dados válidos e consistentes, ou seja, que não haverá dados antigos evitando o problema de atualização perdida [18].

Disponibilidade: Disponibilidade significa que o sistema está disponível para ser acessado no momento em que ele é requisitado. Ou seja, um sistema que está ocupado, sem comunicação ou que não está respondendo, não está disponível [18].

Tolerância ao particionamento: O processamento paralelo e a escalabilidade horizontal são métodos comprovados que estão sendo adotadas como modelo para escalabilidade e aumento de desempenho, em oposição a ampliação e a construção de super computadores.

Os últimos anos têm mostrado que a construção de grandes computadores monolíticos é cara e impraticável na maioria dos casos. Adicionar um número de unidades de hardware em um *cluster* e fazê-las trabalhar em conjunto é uma solução mais econômica e eficiente. O surgimento da computação em nuvem é um testemunho desse fato [18]. Particionamento e falhas ocasionais em um *cluster* são esperadas por causa da escalabilidade horizontal. O terceiro pilar do teorema CAP consiste na tolerância ao particionamento ou tolerância a falhas. A tolerância ao particionamento mensura a capacidade de um sistema continuar disponível, mesmo que alguma parte do *cluster* fique indisponível [18].

Capítulo 3

Cassandra

Neste capítulo, é abordado o banco de dados Cassandra. Na Seção 3.1, são feitas algumas definições e descrição das principais características do Cassandra. Na Seção 3.2, o modelo de dados usado no Cassandra é apresentado de forma breve, trazendo conceitos de: *cluster*, *keyspace*, família de colunas e colunas. Na Seção 3.3, a arquitetura do banco é abordada, explicando sobre: como é feita a comunicação entre nós, como os dados são distribuídos e replicados no ambiente, quais são os particionadores disponíveis para o uso, o que são *snitches* e como funciona solicitações de clientes. Este capítulo usou a documentação oficial do Cassandra como referência, o Apache Cassandra^{TM} 2.0 Documentation [1].

3.1 Definição

Apache Cassandra é um banco de dados de código aberto NoSQL altamente escalável e é indicado para gerenciar grandes quantidades de dados estruturados, semi-estruturados ou não estruturados em vários *datacenters* e em nuvem. Este banco oferece disponibilidade contínua e escalabilidade linear através de muitos servidores sem ponto de falha, juntamente com um modelo de dados dinâmico projetado para flexibilidade e tempos de resposta rápidos [8].

Cassandra fornece uma distribuição automática de dados em todos os nós que participam do anel ou *cluster*, os dados são transparentemente divididos em todos os nós. Cassandra também oferece replicação personalizável, que armazena cópias redundantes de dados entre os nós que participam do anel. Isto significa que se algum nó em um *cluster* ficar fora de operação, uma ou mais cópias dos dados deste nó ficará disponível em outros nós do *cluster*. A replicação pode ser configurada para trabalhar em vários *datacenters* e em múltiplas zonas de disponibilidade na nuvem. Cassandra fornece escalabilidade linear, significando que a capacidade poder ser incrementada adicionando novos nós *online* [8].

Muitas empresas têm implantado o Apache Cassandra, incluindo: Adobe, Comcast, eBay, Rackspace, Netflix, Twitter e Cisco. O maior ambiente de produção têm centenas de *terabytes* de dados em *clusters* com mais de 300 servidores [8].

Algumas das principais características do Cassandra [8] são:

- Escalabilidade elástica: Permite facilmente adicionar capacidade *online* para acomodar mais clientes e mais dados sempre que necessário.

- Arquitetura disponível: Projetado para não contém pontos de falha, resultando em disponibilidade contínua para aplicações críticas que não podem ficar inoperantes.
- Rápido desempenho linearmente escalar: Permite respostas rápidas com escalabilidade linear para responder os clientes.
- Armazenamento de dados flexível: Acomoda facilmente toda a gama de formatos de dados, incluindo: dados estruturados, semi-estruturados e não estruturados, que são executados através de aplicações modernas. Também permite alterações dinâmicas nas estruturas dos dados, de acordo com a necessidade de evolução destes dados.
- Fácil distribuição de dados: Dá a máxima flexibilidade para distribuir os dados onde for necessário por meio da replicação de dados em vários *datacenters*, nuvens e até mesmo nuvens mistas.
- Simplicidade operacional: Com todos os nós do *cluster* tendo os mesmos dados, não há nenhuma configuração complexa para gerenciar, tornando a administração muito simplificada.
- Gerenciamento de Transação: Oferece atomicidade, isolamento e durabilidade das propriedades ACID através da utilização do *commit log* para capturar todas as escritas e redundâncias que garantem a durabilidade dos dados em caso de falhas de hardware.

3.2 Modelo de Dados

O Cassandra é baseado no modelo de dados orientado a coluna. Sua estrutura é formada por um *keyspace* contendo famílias de colunas, que por sua vez é um conjunto de linhas englobando várias colunas. A seguir cada parte deste estrutura é explicado.

3.2.1 *Cluster*

Cassandra é projetado especificamente para ser distribuído em várias máquinas operando em conjunto, de forma invisível para o usuário final, parecendo ser uma única máquina. A estrutura mais externa do Cassandra é o *cluster*, às vezes chamado de anel, pois os nós do *cluster* são organizados em forma de anel. Um nó mantém uma réplica para diferentes intervalos de dados, de forma que se um nó ficar inoperante, uma réplica em outro nó poderá responder às consultas. O protocolo *peer-to-peer* permite que os dados se repliquem entre os nós de forma transparente para o usuário, utilizando o fator de replicação, que é o número de máquinas do *cluster* que irão receber cópias dos mesmos dados [14].

3.2.2 *Keyspace*

Um *cluster* contém *keyspaces*, tipicamente um único *keyspace*. *Keyspace* é a estrutura mais externa para os dados do Cassandra e cada um destes recebe um nome e um conjunto de atributos que definem o comportamento de todo o *keyspace*. Embora muitas vezes é aconselhado criar um único *keyspace* por aplicação, em alguns caso é aceitável e

perfeitamente possível criar quantos *keyspaces* forem necessários para a aplicação. Dependendo das restrições de segurança e do particionador, não há problemas em executar vários *keyspaces* no mesmo *cluster* [14]. Na Figura 3.1, o *keyspace* é simbolizado pelo maior retângulo intitulado "KeySpace 1".

Alguns atributos que podem ser definidos no *keyspace* são [14]:

- Fator de replicação: Refere-se ao número de nós que irão manter cópias ou réplicas de cada linha de dados, essa replicação é transparente ao cliente. O fator de replicação essencialmente permite decidir o quanto se paga em desempenho para ganhar em consistência. Ou seja, o nível de consistência para a leitura e a escrita de dados é baseado no fator de replicação.
- Estratégias de réplicas: Refere-se a forma como as réplicas serão colocados no anel. Existem diferentes estratégias para determinar quais nós vão ter cópias de quais chaves.
- Família de Colunas: Um *keyspace* contém um conjunto de famílias de colunas. Família de colunas contém uma coleção de linhas, cada linha contém colunas ordenadas. Famílias de colunas representam a estrutura dos dados. Cada *keyspace* tem pelo menos uma família de colunas.

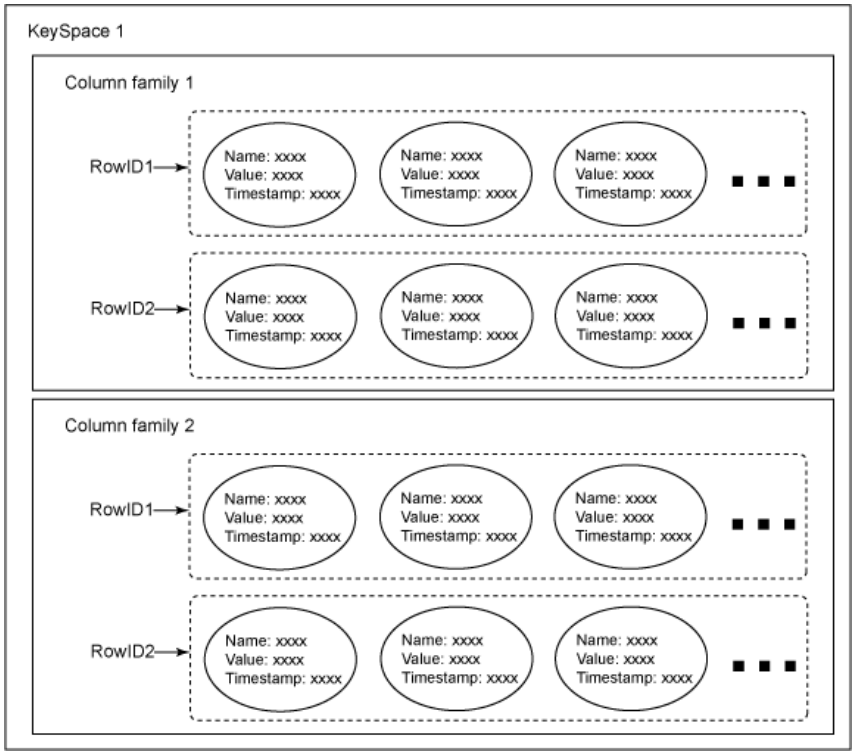

Figura 3.1: Exemplo de modelo de dados Cassandra [2]

3.2.3 Família de Colunas

A família de colunas armazena uma coleção ordenada de linhas de dados, cada uma das quais é uma coleção ordenada de colunas. Cassandra é considerado livre de esquema, pois embora as famílias de colunas sejam definidas, as colunas não são. Pode-se adicionar livremente qualquer coluna em qualquer família de colunas a qualquer momento, dependendo das necessidades. Uma família de colunas tem dois atributos: um nome e um comparador. O comparador indica como as colunas serão classificadas quando são devolvidas ao cliente em uma consulta, podendo ser *long*, *byte*, UTF8, dentre outros [14].

Ao escrever dados em uma família de colunas, deve ser especificado valores para uma ou mais colunas. Esse conjunto de valores, juntamente com um identificador único é chamado de linha [14]. Este identificador único é a chave primária, que pode ser formada por um conjunto de colunas. O primeiro componente de uma chave primária de uma família de colunas é a chave de partição. Dentro de um particionamento, as linhas são agrupadas pelo restante das colunas que compõe a chave primária. As outras colunas são indexados separadamente a partir da chave primária. As famílias de colunas podem ser criadas, excluídas e alterados em tempo de execução, sem bloquear atualizações e consultas [3].

Na Figura 3.1, a família de colunas é representada pelos retângulos intitulados "Column family 1 "e "Column family 2", onde estão contidas as linhas, que são representadas pelos retângulos pontilhados e as setas que apontam para as linhas ilustram a chave de partição.

3.2.4 Coluna

A coluna é a unidade mais básica da estrutura de dados no modelo de dados do Cassandra. A coluna é formada por um nome, um valor e um *timestamp* [14]. Na Figura 3.1, as colunas são representadas pelos círculos.

3.3 Arquitetura

Cassandra foi projetado para lidar com grandes quantidades de dados em vários nós sem nenhum ponto de falha. Sua arquitetura considera que falhas de sistema e de hardware podem acontecer. Assim, Cassandra aborda o problema de falhas através do emprego de um sistema distribuído *peer-to-peer*, onde todos os nós são iguais e os dados são distribuídos entre todos os nós do *cluster*. Cada nó troca informações com o *cluster* o tempo todo. Uma escrita sequencial grava dados no *commit log* de cada nó que captura a atividade de escrita para garantir a durabilidade dos dados. Os dados também são escritos em uma estrutura na memória chamada de *memtable*, que se assemelha a um *cache write-back*. Uma vez que a estrutura de memória estiver cheia, os dados são gravados no disco em um arquivo de dados chamado *sstable*. Todas as escritas são automaticamente particionadas e replicadas em todo o *cluster*. Usando um processo chamado de compactação, Cassandra consolida periodicamente *sstables*, descarta os *tombstones*, que são indicadores de que uma coluna foi excluída e regenera o índice no *sstable* [1].

Arquitetura do Cassandra permite que qualquer usuário autorizado se conecte a qualquer nó em qualquer *datacenter* e utilize a linguagem CQL. Para facilidade de uso, CQL

utiliza uma sintaxe semelhante ao SQL. Do ponto de vista CQL, o banco de dados é estruturado em tabelas. Os desenvolvedores podem acessar o CQL através cqlsh, bem como por meio de *drivers* para linguagens de aplicação [1].

As solicitações do cliente para ler ou escrever podem ir para qualquer nó no *cluster*. Quando um cliente se conecta a um nó com uma solicitação, este nó funciona como coordenador para o operação do cliente particular. O coordenador age como um *proxy* entre a aplicação cliente e os nós que possuem os dados que estão sendo solicitados. O coordenador determina quais os nós do anel que devem atender o pedido com base na configuração do *cluster* [1].

Alguns conceitos importantes para o entendimento desta Seção:

- *Datacenter*: Um grupo de nós relacionados configurados em conjunto dentro de um *cluster* para fins de replicação e carga de trabalho de segregação. Não é necessariamente um *datacenter* físicos. Dependendo do fator de replicação, os dados podem ser gravados em vários *datacenters*.
- *Commit log*: Todos os dados são gravados primeiramente em um log para garantir a durabilidade. Depois que todos os dados forem liberados para *sstables*, o *log* pode ser arquivado, apagado ou reciclado.
- *Sstable*: É um arquivo imutável de dados que o Cassandra usa para escrever os *memtables* periodicamente. *Sstable* apenas anexa e armazena sequencialmente dados no disco para cada tabela.
- *Memtable*: É um *cache write-back* de partições de dados organizados por chave.
- *Rack*: É um armário utilizado para armazenar os equipamentos computacionais de um servidor.

3.3.1 Comunicação entre Nós

Cassandra usa um protocolo chamado *gossip* para descobrir a localização e informações dos estados dos outros nós que participam do *cluster*. *Gossip* é um protocolo de comunicação *peer-to-peer* onde os nós trocam periodicamente informações dos estados deles mesmos e dos outros nós que pertencem ao *cluster*. Assim, todos os nós aprendem rapidamente sobre todos os outros nós do *cluster*. O processo *gossip* troca mensagens de estado com no máximo três outros nós do *cluster*. Uma mensagem *gossip* tem uma versão associada, assim durante a troca de mensagens, as informações mais antigas são substituídas pelas informações mais recentes sobre o estado mais atual de um nó particular [1].

Quando um nó é iniciado pela primeira vez, seu arquivo de configurações determinará qual é o nome do *cluster* que o nó pertence. Além disso, informará com quais nós *seeds* (nós que mantém informações sobre todos os outros nós do *cluster*) devem ser feitos contatos. E determinará outros parâmetros sobre informações de porta e faixas [1].

As informações *gossip* são mantidas localmente por cada nó para serem usadas imediatamente quando o nó é reiniciado. Assim não é necessário esperar uma comunicação *gossip* para obter informações [1].

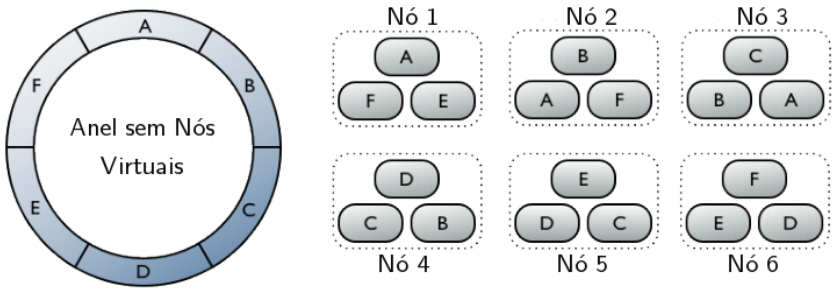

Figura 3.2: Anel sem Nós Virtuais [1]

3.3.2 Distribuição de Dados e Replicação

Distribuição de dados e replicação são conceitos que estão interligados, pois o Cassandra é um sistema *peer-to-peer* que faz cópias de dados e distribui essas cópias entre um grupo de nós. Os dados são organizados por tabela e identificados por uma chave primária. A chave primária determina em qual nó os dados serão armazenados. Cópias de linha de dados são chamadas de réplicas [1].

Mecanismo de *hashig* consistente: O mecanismo de *hashing* consistente distribui dados em um *cluster*. Partições de dados *hashing* consistentes se baseiam na chave de partição. Cassandra atribui um valor de *hash* para cada chave de partição e cada nó do *cluster* fica responsável por uma faixa de valores de *hash*, assim cada nó fica responsável por um conjunto de valores baseados no *hash*. Cassandra coloca os dados em cada nó de acordo com o valor da chave de partição e a faixa que o nó é responsável [1].

Nó virtual: Antes da versão 1.2 do Cassandra, era necessário calcular e atribuir um único *token* para cada nó do *cluster*. Cada *token* determina a posição do nó no anel e a sua porção de dados de acordo com o valor *hash*. Embora o *hashing* consistente fosse usado antes da versão 1.2, era necessário um esforço substâncial para mover um único valor do nó quando nós eram adicionados ou removidos do *cluster* [1].

Cassandra mudou esse paradigma de um *token* e uma faixa por nó para vários *tokens* por nó. O novo paradigma chamado de nó virtual permiti que cada nó possua um grande número de pequenas faixas de partição distribuídas por todo o *cluster*. Nós virtuais também utilizam *hashing* consistente para distribuir dados [1].

A Figura 3.2 mostra um *cluster* sem nós virtuais. Neste paradigma, para cada nó é atribuído um único *token* que representa um local no anel. A escolha do local que uma linha ficará é determinada pelo mapeamento da chave de partição para um valor de *token* dentro de uma faixa a partir do nó anterior até o valor designado. Cada nó também contém cópias de cada linha de outros nós do *cluster*. Por exemplo, a faixa E replica os nós 5, 6 e 1. Um nó possui exatamente uma faixa de partição que é contígua no anel. Por exemplo, a faixa A, F, E é atribuída para o nó 1, uma faixa contígua no anel [1].

A Figura 3.3 mostra um anel com nós virtuais. Dentro de um *cluster*, os nós virtuais são selecionados aleatoriamente e de forma não contígua. A escolha do local que uma linha

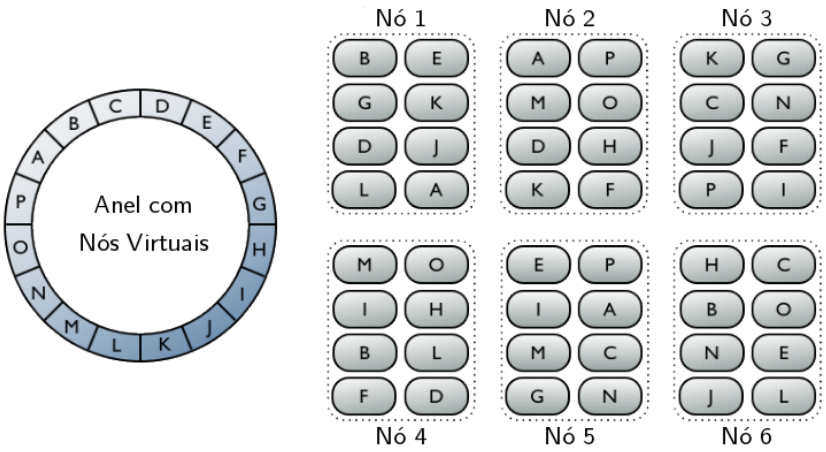

Figura 3.3: Anel com Nós Virtuais [1]

ficará é determinada pelo *hash* da chave de partição dentro de várias partições menores que pertencem a cada nó [1].

Replicação de dados: Cassandra faz o armazenamento de réplicas em vários nós para garantir confiabilidade e tolerância a falhas. A estratégia de replicação determina em quais nós serão colocadas as réplicas [1].

O número total de réplicas no *cluster* é determinado pelo fator de replicação. Um fator de replicação com valor 1 quer dizer que existe apenas uma cópia de cada linha em um nó. Um fator de replicação igual a 2 significa duas cópias de cada linha em diferentes nós. Todas as réplicas são igualmente importantes, não há hierarquia entre réplicas. Como regra geral, o fator de replicação não deve ser maior que o número de nós do *cluster*. Quando o fator de replicação é superior ao número de nós, as escritas são rejeitadas, mas as leituras são permitidas enquanto o nível de consistência desejada puder ser satisfeito [1].

A estratégia de replicação chamada *SimpleStrategy* é recomendada quando o *cluster* é implantado em apenas um *datacenter*. Esta estratégia coloca a primeira réplica em um nó determinado pelo particionador e as réplicas adicionais são colocadas nos próximos nós no sentido horário no anel sem considerar a topologia [1].

A estratégia de replicação chamada *NetworkTopologyStrategy* é recomendada quando o *cluster* é implantado em vários *datacenters*. Esta estratégia especifica quantas réplicas serão colocadas em cada *datacenter*. As réplicas são colocadas no mesmo *datacenter* caminhando no anel no sentido horário até chegar no primeiro nó de outro *rack*. *NetworkTopologyStrategy* tenta colocar réplicas em *racks* distintos, porque nós no mesmo *rack* (ou no mesmo agrupamento físico) podem ficar indisponíveis ao mesmo tempo devido problemas de energia, refrigeração ou rede [1].

3.3.3 Particionador

Um particionador determina como os dados serão distribuídos entre os nós do *cluster*, incluindo réplicas. Basicamente, um particionador é uma função *hash* que calcula o *token*

de uma chave de partição. Cada linha de dados é identificada por uma única chave de partição e distribuída no *cluster* pelo o valor do *token* [1].

Cassandra oferece três particionadores: *Murmur3Partitioner*, *RandomPartitioner* e o *ByteOrderedPartitioner* [1].

Tanto o *Murmur3Partitioner*, como o *RandomPartitioner* utilizam *tokens* para ajudar a distribuir porções iguais de dados em cada nó e distribuir uniformemente dados de todas as tabelas ao longo do anel ou de outro agrupamento, tal como um *keyspace*. Além disso, as solicitações de leitura e escrita também são distribuídas de forma uniforme e as cargas são balanceadas e simplificadas, pois cada parte da faixa de *hash* recebe em média um número igual de linhas [1].

No particionador *Murmur3Partitioner*, os dados são distribuídos uniformemente no *cluster* com base nos valores de uma função *hash*. Este é o particionador padrão, fornece um desempenho melhor em comparação como o particionador padrão anterior, *Random-Partitioner*. O *Murmur3Partitioner* usa a função de *hash MurmurHash*, essa função cria um valor de 64 bit para a chave de partição [1].

No particionador *RandomPartitioner*, os dados são distribuídos uniformemente no *cluster* com base nos valores de uma função *hash*. Este particionador era o padrão antes de Cassandra 1.2, porém ainda pode ser utilizado. O *RandomPartition* distribui dados entre os nós usando a função de *hash* MD5 [1].

No particionador *ByteOrderedPartitioner*, os dados são distribuídos no *cluster* em ordem alfabética. É utilizado quando se deseja um particionamento ordenado. Este particionador ordena as linhas em ordem alfabética. Os *tokens* são calculados pela representação hexadecimal dos caracteres iniciais da chave de partição [1].

3.3.4 *Snitch*

O *snitch* determina a partir de quais *datacenters* e *racks* serão lidos e escritos dados. O *snitch* informa sobre a topologia da rede, de modo que as solicitações são encaminhadas de forma eficiente e permite distribuir réplicas por agrupamento de máquinas em *datacenters* e *racks*. Todos os nós devem ter exatamente a mesma configuração de *snitch*. Cassandra faz o possível para não manter mais do que uma réplica no mesmo *rack*, que não é necessariamente um local físico [1].

Propriedade *snitching* dinâmico: Monitora o desempenho de leituras das réplicas e escolhe a melhor réplica com base no histórico. Por padrão, todos os *snitches* usam uma camada *snitching* dinâmico que monitora a latência de leitura e roteia as solicitações para longe dos nós com desempenho ruim, sempre quando possível. O *snitching* dinâmico é ativado por padrão e é recomendado para uso na maioria das implementações [1].

Abaixo são listados os tipos de *snitches* disponíveis [1]:

- *SimpleSnitch*: Este é o *snitch* padrão, ele não reconhece informações de *datacenters* ou *racks* e é usado em implementações de *datacenter* único.
- *RackInferringSnitch*: Este *snitch* determina a localização dos nós pelo endereço IP, que assume que o 3º octeto do IP corresponde ao rack e 2º octeto ao *datacenter*.
- *PropertyFileSnitch*: Este *snitch* determina a localização dos nós usando um arquivo de configurações definido pelo usuário, que contém os detalhes da rede.

- *GossipingPropertyFileSnitch*: Este *snitch* determina o *datacenter* e *rack* do nó local e usa *gossip* para propagar essa informação para os outros nós.
- *EC2Snitch*: Este *snitch* é usado para implementações utilizando Amazon EC2, onde todos os nós do *cluster* estão dentro de uma única região.
- *EC2MultiRegionSnitch*: Este *snitch* é usado para implementações utilizando Amazon EC2, onde um *cluster* abrange várias regiões.

3.3.5 Solicitação do Cliente

As solicitações de leitura e escrita do cliente podem ir para qualquer nó do *cluster*, porque todos os nós do Cassandra são *peers*. Quando um cliente se conecta a um nó e emite um pedido de leitura ou escrita, esse nó funciona como coordenador para o operação do cliente específico. O trabalho do coordenador é atuar como um *proxy* entre o aplicativo cliente e os nós ou réplicas que possuem os dados que estão sendo solicitados. O coordenador determina quais os nós do anel devem atender ao pedido baseado nas configurações do particionador e estratégia de réplicas do *cluster* [1].

Nível de Consistência de Escrita

O nível de consistência de escrita especifica quantas réplicas devem fazer a escrita de forma bem sucedida antes de enviar uma confirmação para o aplicativo cliente [1].

- ANY: A escrita deve ser realizada por pelo menos um nó. Se todos os nós das réplicas ficarem indisponíveis no momento da escrita, qualquer escrita fica ilegível até que os nós se recuperarem.
- ALL: A escrita deve ser realizada no *commit log* e na *memtable* de todos os nós das réplicas do *cluster* para a linha de dados.
- EACH_QUORUM: A escrita deve ser realizada no *commit log* e na *memtable* em um quorum de nós das réplicas em todos os *datacenters*.
- LOCAL_ONE: A escrita deve ser enviada para pelo menos um nó da réplica do *datacenter* local e ser bem sucedida.
- LOCAL_QUORUM: A escrita deve ser realizada no *commit log* e na *memtable* em um quorum de nós das réplicas em um mesmo *datacerter* e no nó coordenador.
- LOCAL_SERIAL: A escrita deve ser condicionalmente realizada no *commit log* e na *memtable* em um quorum de nós das réplicas em um mesmo *datacerter*.
- ONE: A escrita deve ser realizada no *commit log* e na *memtable* por pelo menos um nó.
- QUORUM: A escrita deve ser realizada no *commit log* e na *memtable* em um quorum de nós das réplicas.
- SERIAL: A escrita deve ser realizada condicionalmente no *commit log* e na *memtable* em um quorum de nós das réplicas.

- THREE: A escrita deve ser realizada no *commit log* e na *memtable* por pelo menos três nós.
- TWO: A escrita deve ser realizada no *commit log* e na *memtable* por pelo menos dois nós.

Nível de Consistência de Leitura

O nível de consistência de leitura especifica quantas réplicas devem responder a uma solicitação de leitura antes de enviar os dados para o aplicativo cliente. Cassandra verifica o número especificado de réplicas para os dados mais recentes com base no *timestamp* para satisfazer o pedido de leitura [1].

- ALL: Retorna o registro com o *timestamp* mais recente, depois que todas as réplicas responderem. A operação de leitura falhará se uma réplica não responder.
- EACH_QUORUM: Retorna o registro com o *timestamp* mais recente, quando um quorum de réplicas de cada *datacenter* do *cluster* responder.
- LOCAL_ONE: Retorna a resposta da réplica mais próxima, determinada pelo *snitch*, mas somente se a réplica estiver no *datacenter* local.
- LOCAL_QUORUM: Retorna o registro com o *timestamp* mais recente, quando um quorum de réplicas do *datacenter* local do *cluster* e o nó coordenador, responderem.
- LOCAL_SERIAL: Permite a leitura do estado atual dos dados sem propor uma nova adição ou atualização. Se uma leitura encontrar uma transação não confirmada em andamento, ela irá confirmar a transação como parte da leitura.
- ONE: Retorna a resposta da réplica mais próxima, determinada pelo *snitch*.
- QUORUM: Retorna o registro com o *timestamp* mais recente, quando um quorum de réplicas independente do *datacenter* responder.
- SERIAL: Permite a leitura do estado atual dos dados sem propor uma nova adição ou atualização. Se uma leitura encontrar uma transação não confirmada em andamento, ela irá confirmar a transação como parte da leitura.
- TWO: Retorna a resposta das duas réplicas mais próxima, determinada pelo *snitch*.
- THREE: Retorna a resposta das três réplicas mais próxima, determinada pelo *snitch*.

Solicitação de Escrita

O coordenador envia uma solicitação de escrita para todas as réplicas que detém a linha que está sendo escrita. Enquanto todos os nós das réplicas estão disponíveis, eles vão receber a escrita independentemente do nível de consistência especificado pelo cliente. O nível de consistência de escrita determina quantos nós devem responder com uma confirmação de sucesso para que a escrita seja considerada bem sucedida. Sucesso significa que os dados foram escritos no *commit log* e no *memtable* [1].

Por exemplo, em uma implementação com um *datacenter* que tem apenas um *cluster*, 10 nós e fator de replicação igual a 3, uma escrita é encaminhada para todos os 3 nós que

detém a linha solicitada. Se o nível de consistência de escrita especificado pelo cliente é ONE, o primeiro nó que completar a escrita responde de volta para o coordenador e então o coordenador envia uma mensagem de sucesso de volta ao cliente. Um nível de consistência igual a ONE significa que é possível 2 das 3 réplicas perderem a escrita, se os nós ficassem inoperantes no momento em que o pedido foi feito. Se uma réplica perde uma escrita, Cassandra faz a linha de dados ficar consistente usando posteriormente um dos seus mecanismos internos de reparação [1].

Em implantações com vários *datacenters*, Cassandra otimiza o desempenho de escrita escolhendo um nó coordenador em cada *datacenter* remoto para lidar com as solicitações de réplicas dentro desse *datacenter*. O nó coordenador acionado pela aplicação cliente precisa encaminhar o pedido de escrita para somente um nó de cada *datacenter* remoto [1].

Se estiver usando um nível de consistência ONE ou LOCAL_QUORUM, somente os nós do mesmo *datacenter* e o nó coordenador devem responder à solicitação do cliente e informar o sucesso. Desta forma, a latência geográfica não afeta o tempo de resposta das solicitações do cliente [1].

Solicitação de Leitura

Existem dois tipos de solicitações de leitura que um coordenador pode enviar para uma réplica, a solicitação de leitura direta e a solicitação de recuperação de leitura em plano de fundo [1].

O número de réplicas contactadas por uma solicitação de leitura direta é determinado pelo nível de consistência especificado pelo cliente. Solicitações de recuperação de leitura em plano de fundo são enviadas para qualquer réplica adicional que não recebeu uma solicitação direta. Solicitações de recuperação de leitura garantem que a linha solicitada é consistente em todas as réplicas [1].

Assim, o coordenador primeiramente entra em contato com as réplicas especificadas pelo nível de consistência. O coordenador envia esses pedidos para réplicas que estão respondendo mais rapidamente. Os nós contactados respondem com os dados solicitados, se vários nós são contactados, as linhas de cada réplica são comparadas na memória para ver se elas são consistentes. Se elas não são, então a réplica que tem dados mais recentes é usada pelo coordenador para transmitir o resultado de volta para o cliente [1].

Para garantir que todas as réplicas tenham a versão mais recente dos dados que são lidos com maior freqüência, o coordenador também entra em contato e compara os dados de todas as réplicas restantes que possuem a linha em plano de fundo. Se as réplicas são inconsistentes, o coordenador envia uma solicitação de escrita para as réplicas atualizarem a linha com os valores mais recentes. Este processo é conhecido como reparação de leitura [1].

Capítulo 4

Estudo de Caso e Implementação

Neste capítulo, é feita uma apresentação do Sistema Integrado de Administração de Recursos Humanos, explicado como ele se insere na administração publica e exposto o modelo de dados utilizado no ambiente PostgreSQL para buscar a trilha de auditoria para incompatibilidade de rubricas. Também é abordado um modelo de dados, bem como as etapas necessárias para implementação e configuração do ambiente Cassandra e aplicações para resolver o problema foco deste trabalho.

4.1 Estudo de Caso

O Sistema Integrado de Administração de Recursos Humanos (SIAPE) é um sistema de abrangência nacional criado para integrar todas as plataformas de gestão da folha de pessoal dos servidores públicos. O SIAPE é um dos principais sistemas que estruturam o governo. Este sistema é o pilar da integração dos órgãos pertencentes ao Sistema de Pessoal Civil da Administração Pública Federal, além de ser responsável pelo envio das informações referentes ao pagamento dos servidores às Unidades Pagadoras dos órgãos. Também garante a disponibilidade desses dados na página SiapeNet, bem como o envio dos arquivos de crédito para os bancos responsáveis pelo pagamento [6].

O siapenet.gov.br é o site oficial das informações do SIAPE. Os servidores, aposentados e pensionistas, que totalizam 1.400.000 pessoas, acessam o sistema de qualquer lugar do mundo e podem consultar dados de sua vida financeira, funcional e pessoal. Atualmente o SiapeNet é usado por 290 órgãos públicos, processa 1.400.000 contracheques, recebe 1.100.000 visitas por mês, tem 20 milhões de páginas visualizadas mensalmente e 900 *gigabytes* de informação trafegada em cada mês [7].

O projeto AUDIR é responsável por fazer as consultas de trilha de auditoria nos dados do SIAPE. Este projeto mantém um banco de dados com os dados do SIAPE implementado em PostgreSQL v8.4. A configuração do hardware utilizado é um Dell Inc. PowerEdge R610 com 2xCPU Xeon 2.80GHz, 32GB RAM e disco rígido de 500GB [15]. Neste trabalho, foi implementado um modelo de dados para o Cassandra, utilizando um *cluster* com 3 nós, cada nó configurado com 2xCPU Xeon 2.80GHz, 8GB RAM e disco rígido de 500GB.

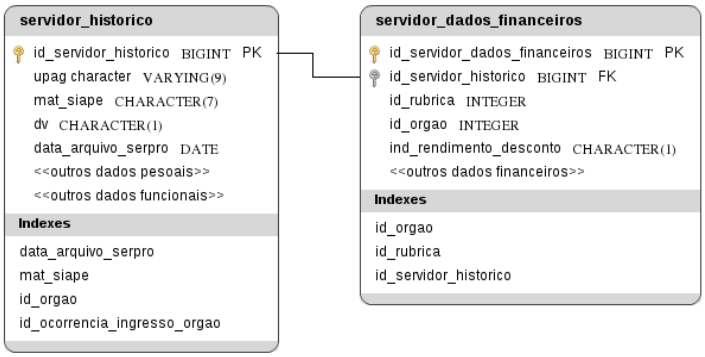

Figura 4.1: Modelo de dados PostgreSQL para dados pessoais, funcionais e financeiros dos servidores [15]

4.2 Modelo de dados PostgreSQL

O arquivo fita espelho tem a gravação sequencial os dados pessoais, funcionais e financeiros dos servidores públicos federais para cada mês. Este arquivo possui informações de trabalhadores, entre eles, ativos, inativos e aposentados. E possui 36 campos de dados pessoais, 153 campos de dados funcionais e 32 campos de dados financeiros, totalizando 221 campos. A estrutura dele utiliza registros de tamanho fixo. Por mês, o tamanho de cada arquivo fita espelho é de cerca de 15GB e tem crescido a cada mês.

O banco de dados mantido pelo AUDIR possui 83 relações, mas somente algumas são importantes para a consulta de trilha de auditoria. As principais relações são: servidor_historico e servidor_dados_financeiros. A relação servidor_historico possui dados pessoais e funcionais dos servidores, 192 colunas e aproximadamente 28.696.200 tuplas. A relação servidor_dados_financeiros possui dados financeiros dos servidores, 20 colunas e aproximadamente 167.077.000 tuplas. A chave primária da relação servidor_historico é chave estrangeira na relação servidor_dados_financeiros. A Figura 4.1 representa a parte do modelo de dados da implementação utilizando PostgrSQL importante para a consulta de trilha de auditoria.

As relações rubrica, rubricas_pontos_controle, rubrica_pc_rubricas_incompativeis e ponto_controle são auxiliares e dizem respeito as rubricas que não podem ser pagas simultaneamente para o mesmo servidor no mesmo mês. A rubrica é uma classificação para os diversos tipos de pagamentos que um servidor pode receber, como por exemplo: pagamento de vencimento básico, pagamento de férias. O ponto de controle é uma classificação para os objetos que devem ser auditados. No caso específico deste trabalho, o foco é o ponto de controle de incompatibilidade de rubricas. A Figura 4.2 representa a parte do modelo de dados que diz respeito as rubricas.

A consulta da trilha de auditoria que busca pagamentos inapropriados para servidores públicos é uma função implementada no PostgreSQL. Tal função busca na relação rubrica_ponto_controle todas as rubricas que têm o identificador de ponto_controle correspondente a incompatibilidade de rubricas. Depois, para cada rubrica x_i encontrada, a função busca na relação rubrica_pc_rubrica_incompatibilidade todas as rubricas que são incompatíveis com a rubrica x_i. Assim, é sabido todas as y_{ij} rubricas que são incompatí-

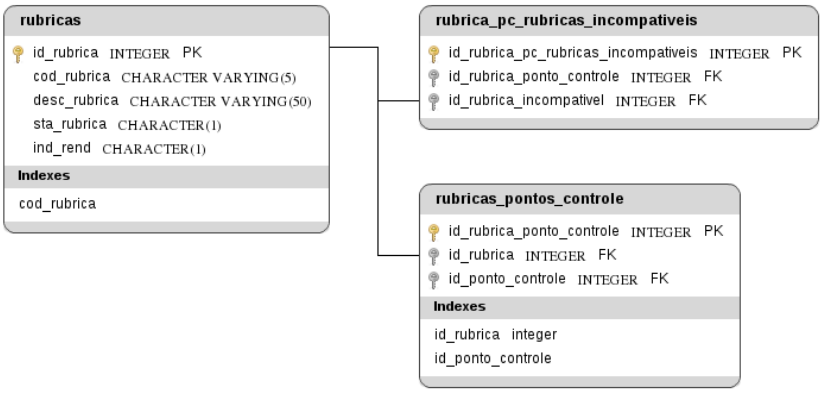

Figura 4.2: Modelo de dados PostgreSQL para rubricas

veis com cada rubrica x_i. É feito, então, um *join* da relação servidor_dados_financeiros com ela mesma para cada rubrica x_i e cada rubrica incompatível y_{ij} para buscar os servidores que recebem as duas rubricas simultaneamente. Nestes *joins* que são feitos, além desta restrição de rubricas, várias outras restrições que levam em conta informações pessoais, funcionais e financeiras para garantir a incompatibilidade. O resultado final é a união dos resultados dos *joins* para cada par de rubricas incompatíveis.

4.3 Modelo de dados Cassandra

O modelo de dados Cassandra é composto pela família de colunas chamada servidor_financeiro que possui apenas a chave primária composta pelas colunas cod_rubrica, ano, mês, cod_orgao, upag e mat_siape, e pela família de colunas chamada rubrica que possui apenas a chave primária composta pelas colunas ponto_controle, cod_rubrica e cod_rubrica_incompativel. A Figura 4.3 representa o modelo de dados da implementação utilizando Cassandra, ele será chamado de Modelo de Dados I.

A consulta por incompatibilidade de rubricas usando o Cassandra busca as rubricas e rubricas incompatíveis basicamente da mesma forma. A grande diferença está em como foi feita a busca dos servidores que recebem cada rubrica, pois o Cassandra não permite *join*. A implementação é detalhada na próxima seção.

Além deste modelo, foi testado um modelo semelhante, mas com índices para cada coluna, os resultados obtidos serão discutidos no Capítulo 5 e este será chamado de Modelo de Dados I com índices.

4.4 Implementação

A implementação proposta neste trabalho se deu em 3 fases: configuração do ambiente, carga de dados e recuperação dos dados para verificar a incompatibilidade de rubricas. Tais fases são explicadas a seguir.

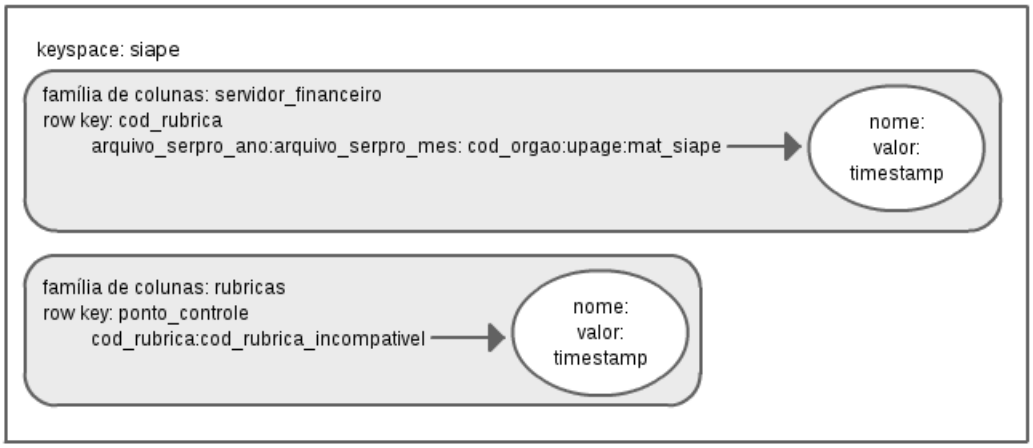

Figura 4.3: Modelo de Dados I

4.4.1 Configurações do Ambiente

O modelo de dados foi implementado em um ambiente formado por um *cluster* com 3 nós. Cada nó foi configurado com 2xCPU Xeon 2.80GHz, 8GB RAM, disco rígido de 500GB e sistema operacional Ubuntu 12.04 LTS.

O Cassandra foi instalado individualmente em cada nó. Foi utilizado a distribuição DataStax Community Edition 2.0.7 do Cassandra, que é uma distribuição gratuita do Apache Cassandra^{TM}disponibilizada pela DataStax. Para a instalação do Cassandra, foi necessário ter as seguintes ferramentas instaladas:

- *Advanced Package Tool* (APT): gerenciador de pacotes do sistema operacional, que auxilia na instalação de pacotes.
- *Oracle Java SE Runtime Environment* (JRE) 7: é um Ambiente de Tempo de Execução Java, utilizado para executar as aplicações da plataforma Java.
- *Java Native Access* (JNA): é uma biblioteca que facilita o acesso aos códigos nativos a partir de aplicações Java.

Após a instalação, foi necessário alterar o arquivo de configurações cassandra.yaml de cada nó para configurar o *cluster*. Foi definido 256 nós virtuais por nó, este valor é o recomendado para manter os nós balanceados. O particionador escolhido foi *Murmur3Partitioner*, pois é o padrão e tem um bom desempenho. O *snitch* definido foi o *SimpleSnitch*, pois a implementação possui apenas um *datacenter*. Também foi escolhido um nó para ser o nó *seed*, ou seja, o nó responsável por manter informações sobre todos os outros nós.

A estrutura do banco de dados foi criada por meio do cqlsh. O cqlsh é um cliente que utiliza linha de comando para executar consultas utilizando a linguagem CQL (Cassandra *Query Language*). Foi criado o *keyspace* com fator de replicação igual a 1 e estratégia de replicação *SimpleStrategy*. As famílias de colunas foram criadas utilizando as configurações padrão do Cassandra, todas as colunas tem comparador inteiro.

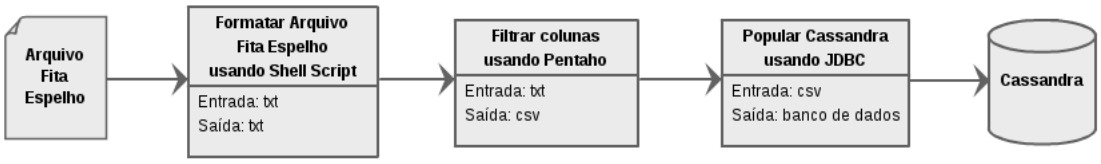

Figura 4.4: Etapas para carga de dados

4.4.2 Carga de Dados no Banco

Para popular o banco de dados Cassandra, foi necessário passar por 3 etapas. A Figura 4.4 ilustra tais etapas. As duas primeiras prepararam os dados para posteriormente carregá-los, de fato, no banco. Foi utilizado a linguagem *Shell Script* para formatar o arquivo fita espelho e, depois, o Pentaho foi utilizado como filtro para as colunas. E finalmente, o JDBC fez a população do banco. Os detalhes de cada etapa são descritos a seguir.

O arquivo formatado do arquivo espelho resulta em um arquivo de texto com 25 colunas necessárias para fazer as restrições que levam em conta informações pessoais, funcionais e financeiras. A linguagem *Shell Script* foi escolhida pela praticidade que ela oferece. Como o arquivo fita espelho utiliza registros de tamanho fixo foi necessário somente percorrer o arquivo texto e retornar algumas colunas.

Ao invés de fazer uma consulta com as restrições diretamente na linguagem de banco de dados, como foi feito no modelo PostgreSQL, foi optado fazer um filtro utilizando o Pentaho, pois o Cassandra não permite consultas muito sofisticadas. o Pentaho Data Integration é uma ferramenta que permite fazer extração, transformação e carregamento de dados. Normalmente chamado de ferramenta ETL que vem do inglês: *Extract Transform Load* [4]. É uma ferramenta fácil de usar, possui uma interface gráfica intuitiva, mas possui limitações. É compatível com Cassandra, possibilitando carregar as familias de colunas diretamente. Porém, não é compatível com o chaves primárias compostas por mais de uma coluna. Por isso, a ferramenta foi utilizada apenas para filtrar e formatar o arquivo. Foi utilizado a versão 4.4.0 do Pentaho Data Integration.

O Pentaho foi utilizado para fazer todas as restrições das informações pessoais, funcionais e financeiras, funcionando como um pre-processamento da consulta de incompatibilidade. O resultado foi um arquivo CSV com 6 colunas e aproximadamente 6.500.000 de linhas. A Figura 4.5 mostra o esquema montado para filtrar as colunas. A quantidade de colunas foi diminuída, pois as restrições das informações pessoais, funcionais e financeiras já foram feitas. Assim, as próximas consultas utilizarão somente a identificação do servidor e as rubricas que ele recebe, sendo necessário apenas a chave primária e a rubrica. O Pentaho foi escolhido pela facilidade de fazer os filtros utilizando interface gráfica. Uma linguagem de programação poderia ter sido utilizada e teria sido muito útil, se os filtros fossem complexos. Mas o Pentaho foi suficiente para faz as restrições, já que as restrições eram somente de valores.

Com o arquivo CSV, as familias de coluna foram populadas utilizando o JDBC. O JDBC é uma API que conecta o Java ao banco de dados, no caso específico conecta-se ao Cassandra. Foi feito um programa em Java utilizando esta API para conectar-se ao Cassandra e popular as familias de colunas percorrendo o arquivo CSV. Foi utilizado

Figura 4.5: Modelo Pentaho

a versão 1.2.5 do Cassandra JDBC e a versão 1.7.0_40 do Java Runtime Environment (JRE). Como o banco já populado, foi possível fazer as consultas para verificar a incompatibilidade de rubricas.

O Cassandra JDBC foi escolhido pela velocidade de inserção dos dados que ele proporcionou. Existem diversos clientes disponíveis que poderiam ter sido utilizados nesta etapa. A inserção de dados utilizando cqlsh, por exemplo, mas este cliente não é otimizado para escrita de muitas linhas, por isso foi descartado. Alguns testes foram feitos com o Cassandra Bulk Loader, que é um cliente que recebe *sstables* já montadas e insere-as no banco, sendo necessário primeiro transformar o arquivo de dados em *sstables* e depois popular o banco. No Capítulo 5, é abordado os resultados encontrados utilizando o Cassandra JDBC e o Cassandra Bulk Loader.

4.4.3 Aplicação Cliente

Foi feita uma aplicação Java utilizando a API Hector para fazer consultas no Cassandra, esta API é um cliente Java de alto nível para Cassandra. Com ela pode ser feito acesso aos dados, como leitura, escrita, atualização e exclusão [5]. Foi utilizado a versão 1.0.2 do Hector Core.

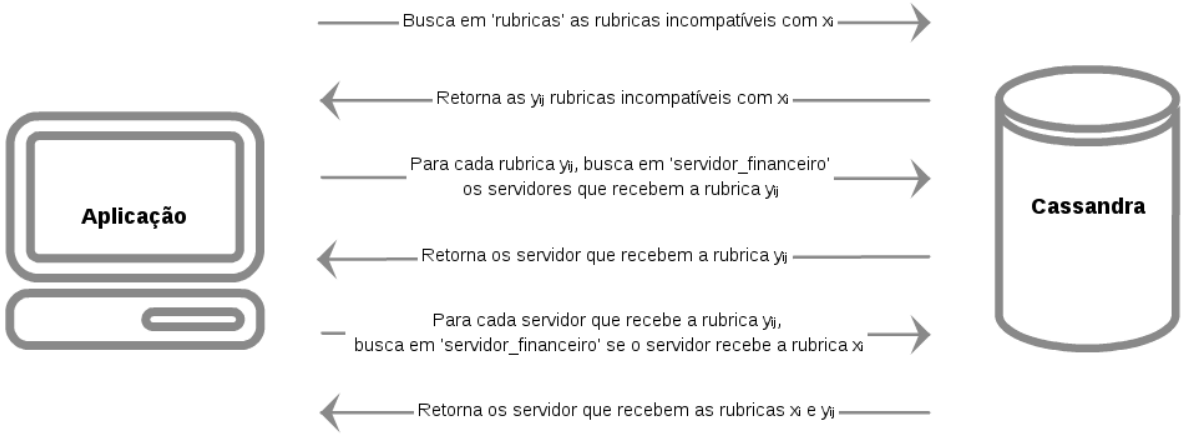

Figura 4.6: Modelo Aplicação

A aplicação busca todas as x_i rubricas que tem o identificador de ponto_controle correspondente a incompatibilidade de rubricas na coluna de famílias rubricas. Para cada rubrica x_i, é criada uma *thread* onde são feitas todas as próximas consultas. Ainda na familia de colunas rubricas, busca-se todas as rubricas incompatíveis com a rubrica x_i.

Para cada rubrica incompatível y_{ij}, são buscados os servidores que recebem a rubrica y_{ij} na familia de colunas servidor_financeiro para o mês e ano definido previamente. Para cada servidor resultante, é feito uma busca na familia de colunas servidor_financeiro para verificar se ele também recebe a rubrica x_i. Se o servidor recebe a rubrica y_{ij} e x_i, ele é classificado como um servidor que recebe incompatibilidade de rubricas. O resultado final é o conjunto de todos os servidores que recebem pelo menos um par de rubricas incompatíveis. A Figura 4.6 ilustra esta busca.

A aplicação também foi testada utilizando somente a principal *thread* da aplicação, os detalhes dos resultados serão abordados no Capítulo 5.

4.5 Dificuldades Encontradas

Inicialmente, foi pensado em um modelo de dados Cassandra que tivesse todos os campos utilizados na consulta de trilha de auditoria para incompatibilidade de rubricas. O modelo de dados foi composto pela família de colunas chamada servidor_financeiro_completo que possui a chave primária composta pelas colunas ano, mês, cod_orgao, upag, mat_siape e cod_rubrica, além das colunas com informações pessoais, funcionais e financeiras importantes para a consulta. Composto também pela família de colunas chamada servidor_financeiro que possui apenas a chave primária composta pelas colunas cod_rubrica, ano, mês, cod_orgao, upag e mat_siape, e pela família de colunas chamada rubrica que possui apenas a chave primária composta pelas colunas ponto_controle, cod_rubrica e cod_rubrica_incompativel. A Figura 4.7 ilustra o modelo.

A aplicação deste modelo buscaria todas as x_i rubricas que tem o identificador de ponto_controle correspondente a incompatibilidade de rubricas na coluna de famílias rubricas. Para cada rubrica x_i, buscaria todas as rubricas incompatíveis com a rubrica x_i na coluna de famílias rubricas. Para cada rubrica incompatível y_{ij}, seriam buscados os servidores que recebem a rubrica y_{ij} na familia de colunas servidor_financeiro para o mês e ano definido previamente. Para cada servidor resultante, seria feito uma busca na familia de colunas servidor_financeiro_completo para verificar se ele também recebe a rubrica x_i. Se o servidor recebesse a rubrica y_{ij} e x_i, ele seria classificado como um servidor que recebe incompatibilidade de rubricas. O resultado final seria o conjunto de todos os servidores que recebem pelo menos um par de rubricas incompatíveis.

Esta aplicação seria bem parecida com a aplicação proposta para o modelo de dados Cassandra descrito na Seção 4.3 porém a ultima consulta, que busca se o servidor que recebem a rubrica y_{ij}, também recebe a rubrica x_i, é feita na familia de colunas servidor_financeiro_completo. Isso foi pensado para que a consulta tirasse proveito da forma que o Cassandra armazena os dados, pois a composição das chaves primárias das familias de colunas servidor_financeiro e servidor_financeiro_completo são diferentes. A familia servidor_financeiro_completo tem a coluna ano como chave de partição, enquanto que a familia servidor_financeiro tem a coluna cod_rubrica como chave de partição. Isso significa que a familia servidor_financeiro_completo armazenaria todos as linhas referentes a um ano especifico em apenas um único nó, enquanto que a familia servidor_financeiro armazenaria todos as linhas referentes a uma rubrica especifica em apenas um único nó.

Este modelo foi implementado e testado, porém obteve resultados ruins. Os resultados desta implementação serão discutidos no Capítulo 5 e será chamado de Modelo de Dados II.

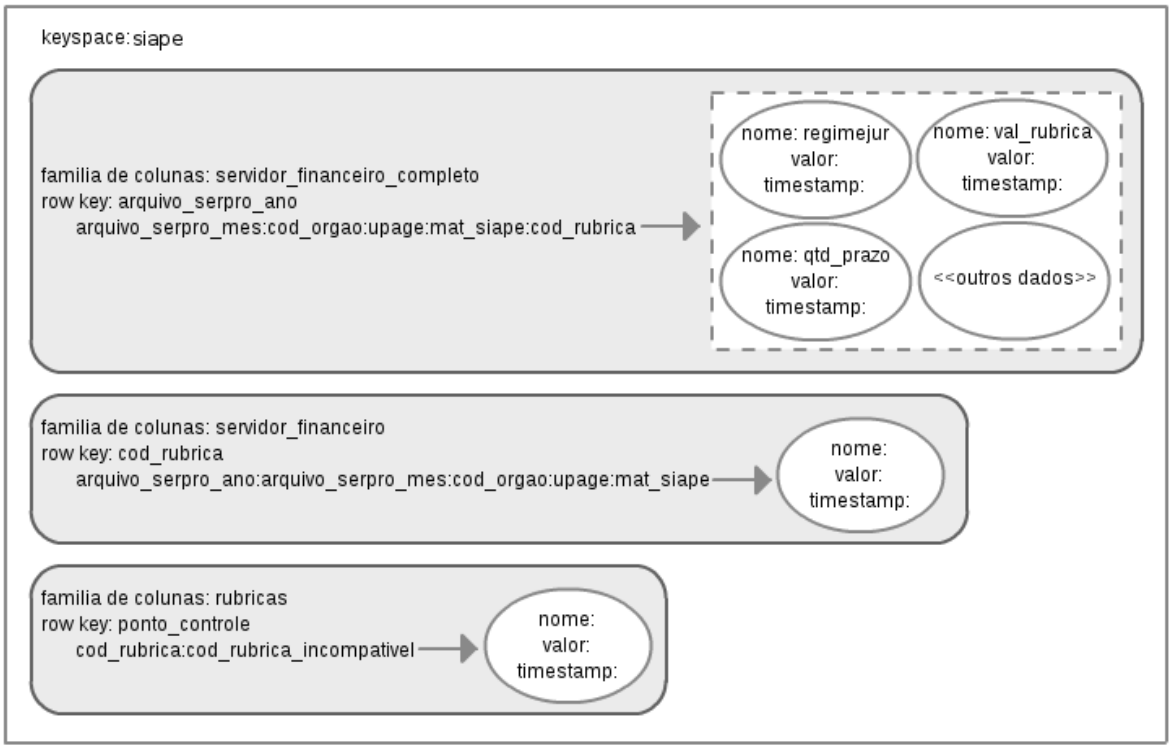

Figura 4.7: Modelo de Dados II

Capítulo 5

Resultados

Neste capítulo, são exibidos os resultados obtidos nos testes de carga de dados utilizando os clientes Cassandra BulkLoader e o Cassandra JDBC, além dos testes da consulta da trilha de auditoria para incompatibilidade de rubricas realizados com cliente Hector com uma única *thread* e com várias *threads*. E é apresentada uma comparação entre o modelo de dados proposto e o real. Os resultados apresentados sobre o modelo de dados real foram retirados de um artigo que faz uma análise entre o uso do Hbase e PostgreSQL para os mesmo dados dos servidores públicos federais utilizados neste trabalho [15].

5.1 Carga de dados

Para a carga de dados, foi feita uma preparação dos dados, que envolveu formatação e filtragem de dados. Depois, os dados foram carregados utilizando dois clientes, o Cassandra BulkLoader e o Cassandra JDBC. E foi feita uma comparação entre os resultados obtidos em relação aos diferentes modelos e entre os clientes. Os detalhes de cada passo estão descritos nesta seção.

5.1.1 Preparação dos Dados

A preparação dos dados consiste nas etapas que antecedem a carga dos dados no Cassandra descritos na Subseção 4.4.2. A etapa que formata a fita espelho utilizando o *Shell Script* tem como resultado um conjunto de dados igual para todos os modelos testados, pois esta etapa apenas diminui a quantidade de colunas da fita. A etapa que filtra colunas utilizando o Pentaho tem com resultado o mesmo conjunto de dados para os dois primeiros modelos descritos na Tabela 5.1, pois a única diferença entre eles é a aplicação de índices. O tempo total gasto pelo Modelo de Dados II foi maior, como esperado, pelo fato dele ter muito mais colunas que os outros.

Tabela 5.1: Preparação dos dados para carga em segundos

	Shell Script	Pentaho	Total
Modelo de Dados I	643s	115s	758s
Modelo de Dados I com Índices	643s	115s	758s
Modelo de Dados II	643s	330s	973s

5.1.2 Testes utilizando Cassandra BulkLoader

Para utilizar o cliente Cassandra BulkLoader, foi necessário primeiramente construir as *sstables* implementando uma aplicação específica para este propósito. Aplicação foi desenvolvida usando a linguagem de programação Java e a API *SSTableSimpleUnsorted-Writer*. Com as *sstables* prontas, foi utilizada uma outra aplicação Java para, de fato, fazer a população de dados no Cassandra. Os tempos gastos para carga de dados estão na Tabela 5.2. A Figura 5.1 exibe graficamente a diferença entre os tempos gastos por cada modelo.

Tabela 5.2: Carga de dados utilizando Cassandra BulkLoader em segundos

	Construir *sstables*	BulkLoader	Total
Modelo de dados I	15912,61s	16,60s	15929,21s
Modelo de dados I com Índices	17315,75s	399,92s	17715,67s
Modelo de dados II	509722,94s	1059,17s	510782,11s

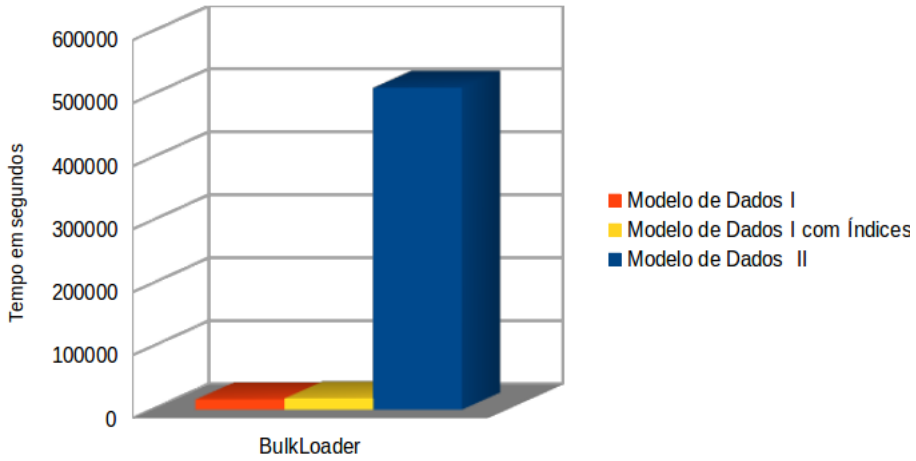

Figura 5.1: Carga de Dados utilizando Cassandra BulkLoader

5.1.3 Testes utilizando Cassandra JDBC

Os tempos gastos para carga de dados utilizando o cliente Cassandra JDBC estão na Tabela 5.3. A Figura 5.2 exibe graficamente a diferença entre os tempos gastos por cada modelo.

Tabela 5.3: Carga de Dados utilizando Cassandra JDBC em segundos

	JDBC
Modelo de dados I	447,22s
Modelo de dados I com Índices	799,41s
Modelo de dados II	4236,05s

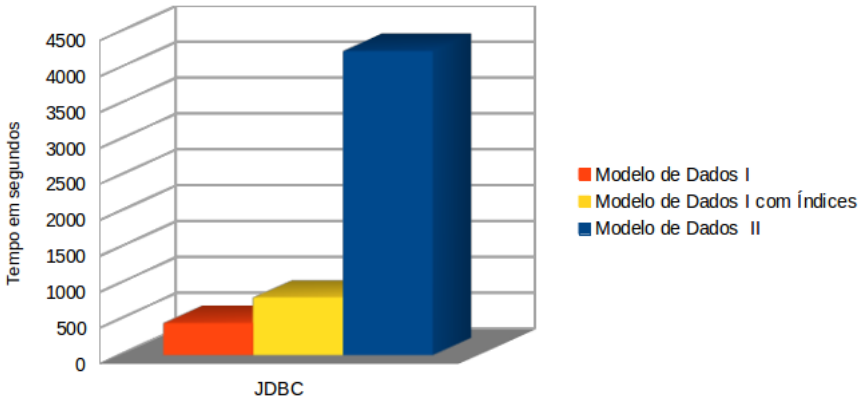

Figura 5.2: Carga de Dados utilizando Cassandra JDBC

5.1.4 Comparação entre modelos

Nos bancos de dados orientados a colunas, a operação de escrita desmembra cada linha em colunas para serem carregadas separadamente [10]. No Cassandra, isso significa que cada linha de dados será dividida pela quantidade de colunas não vazias, e cada coluna será inserida separadamente indexada pela chave primária. Por isso, o Modelo de Dados II gastou mais tempo que os outros modelos para os dois clientes testados, ele tinha mais colunas em sua estrutura. O Modelo de Dados I teve um desempenho melhor para a carga que o Modelo de Dados I com Índices, pois este teve um incremento em seu processamento para fazer a indexação para cada coluna.

5.1.5 Comparação entre clientes

A Figura 5.3 exibe graficamente a diferença entre os tempos gastos por cada cliente para carga de dados dos modelos. O Cassandra JDBC obteve resultado melhor, em comparação com o Cassandra BulkLoader. Este teve tempos muito ruins, pois o gasto para construir as *sstables* foi muito grande.

5.2 Consulta de Incompatibilidade de Rubricas

A consulta da trilha de auditoria para incompatibilidade de rubricas foi testada utilizando o cliente Hector de duas formas diferentes, uma utilizando várias *threads* e outra utilizando somente a *thread* principal. A Tabela 5.4 exibe os resultados obtidos em ambas consultas para as três modelagens.

Tabela 5.4: Consulta utilizando Hector em segundos

	Hector	Hector com *Threads*
Modelo de dados I	3932,18s	842,16s
Modelo de dados I com Índices	5017,66s	1132,05s
Modelo de dados II	5649,97s	2052,05s

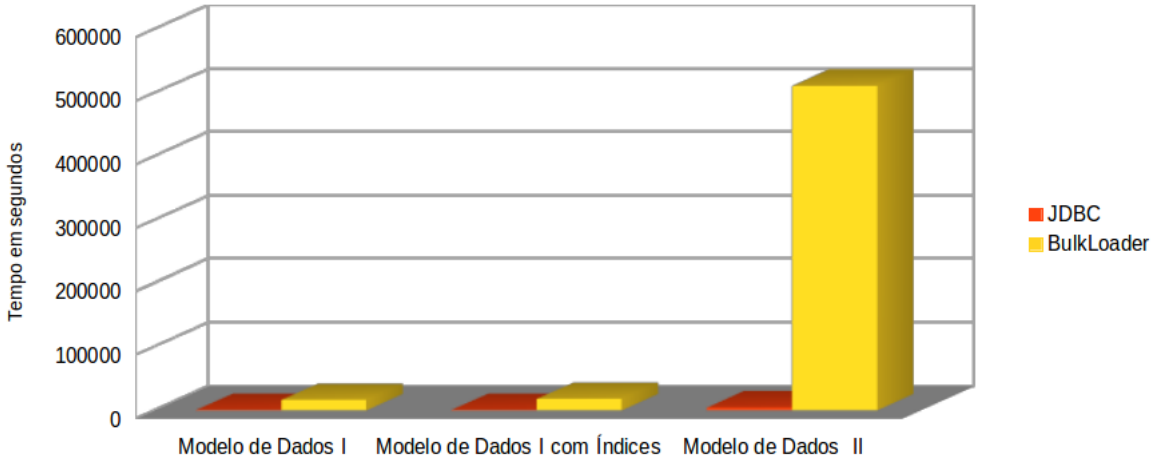

Figura 5.3: JDBC versus BulkLoader

5.2.1 Testes utilizando Hector

A Figura 5.4 apresenta um comparativo entre os modelos de dados para a consulta de incompatibilidade de rubricas utilizando o cliente Hector utilizando somente a *thread* principal.

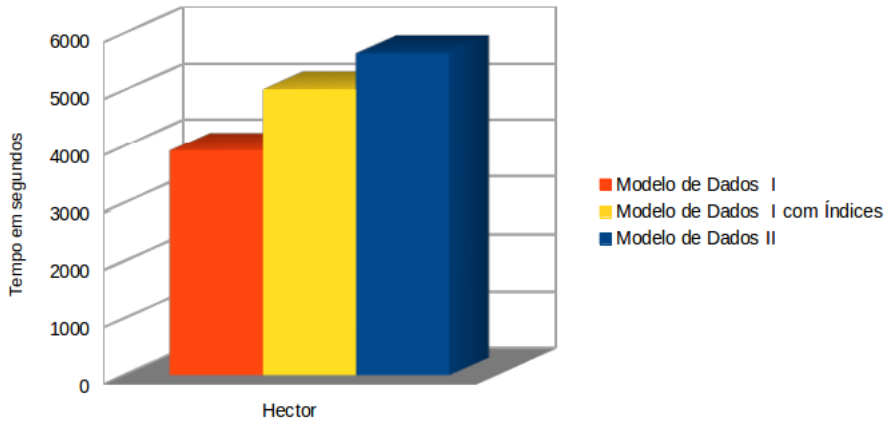

Figura 5.4: Consulta utilizando Hector

5.2.2 Testes utilizando Hector com *Threads*

A Figura 5.5 apresenta um comparativo entre os modelos de dados para a consulta de incompatibilidade de rubricas utilizando o cliente Hector com o uso de várias *threads*.

5.2.3 Comparação

A Figura 5.6 apresenta graficamente uma comparação entre as consultas utilizando o cliente Hector com um e múltiplas *threads*. Era esperado que o uso de *threads* fosse

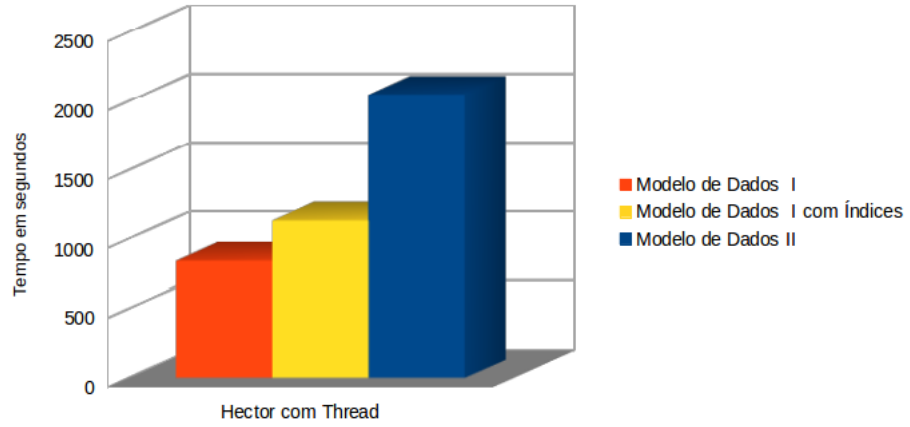

Figura 5.5: Consulta utilizando Hector com *Threads*

consideravelmente mais rápido, assim como foi obtido nos teste. O seu uso possibilitou tirar vantagem do ambiente da aplicação do Cassandra que é composto por três nós.

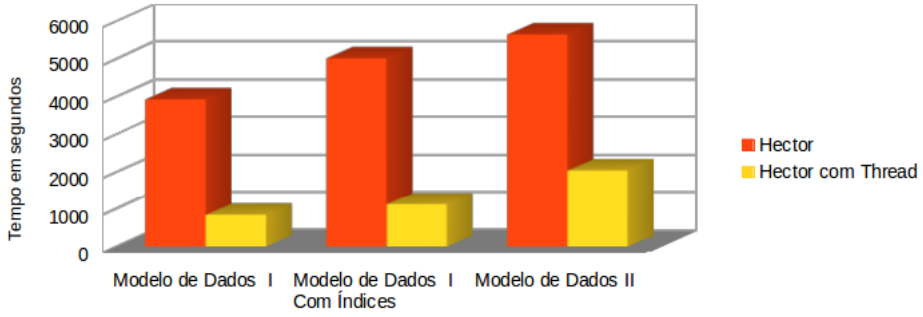

Figura 5.6: Hector versus Hector com *Threads*

5.3 Comparativo entre Modelo Real e Modelo Proposto

A Tabela 5.5 expõe a comparação para a consulta da trilha de auditoria para incompatibilidade de rubricas entre o modelo de dados real implementado em PostgreSQL e o melhor modelo de dados proposto neste trabalho, o Modelo de Dados I que obteve melhor tempo de carga e consulta. A Figura 5.7 apresenta de forma gráfica o comparativo entre os dois modelos.

Tabela 5.5: Consulta: PostegreSQL versus Cassandra

	Consulta de incompatibilidade de rubricas
Modelo Real [15]	1599,96 s
Modelo Proposto	842,16 s
Eficiência	47,36%

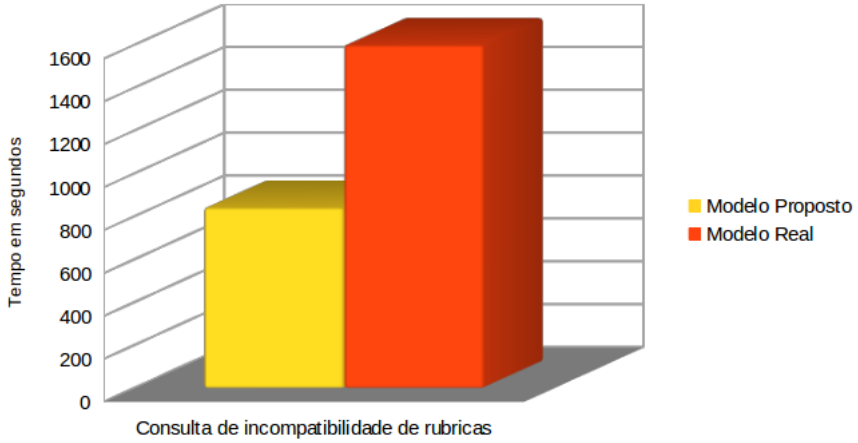

Figura 5.7: Comparativo entre Modelos

O Modelo Proposto teve eficiência de 47,36% em relação ao Modelo Real, isso se deu pela forma que o Cassandra distribui, replica e particiona os dados no *cluster*, permitindo a utilização de uma arquitetura projetada para lidar com paralelismo. A utilização de *theads* na consulta possibilitou usufruir desta estrutura, pois as solicitações de leitura e escrita poderiam ir para qualquer nó do *cluster* sem utilizar concorrência, mas quando um cliente se conecta a um nó e faz uma solicitação, este nó funciona como coordenador da operação atuando como *proxy* e determina quais os nós que devem atender ao pedido baseado nas configurações do particionador e estratégias de replicação do *cluster* [1].

A inserção de dados não foi comparada com o modelo de dados real, pois o modelo proposto foi projetado para resolver o problema especifico da consulta da trilha de auditoria e não persiste todos os dados existentes no modelo real, sendo assim, injusto fazer tal confronto.

Capítulo 6

Conclusões e Trabalhos Futuros

Nesta monografia, foi feito um estudo de casos para analisar os dados dos servidores públicos federais utilizando um banco de dados NoSQL Cassandra. Esta análise consistiu em criar e manter um ambiente Cassandra, escolher ferramentas adequadas, criar modelos de dados, testar a eficiência da inserção e busca de dados e comparar a velocidade da consulta da trilha de auditoria de incompatibilidade de rubricas com o modelo de dados PostgreSQL.

Foram apresentados três modelos de dados diferentes. A carga de dados foi testada para cada um utilizando dois clientes: Cassandra JDBC e Cassandra BulkLoader. O cliente Cassandra JDBC obteve resultado melhor, pelo fato do Cassandra BulkLoader necessitar da construção de *sstables* antes de inserir os dados de fato. Além disso, a consulta da trilha de incompatibilidade de rubricas foi testada fazendo uso do cliente Hector de duas forma: utilizando apenas a *thread* principal da aplicação e utilizando várias *threads*. O uso do Hector com várias *threads* obteve resultado melhor que somente com uma, pois tirou proveito do paralelismo oferecido pelo Cassandra. Foi feita a comparação do modelo proposto, que obteve melhores resultados de carga e busca, com o modelo PostgreSQL. Somente a consulta de incompatibilidade de rubricas foi analisada, pois os bancos de dados orientados a colunas são modelados baseados nas consultas que se deseja fazer. A inserção de dados não foi comparada com o modelo real, pois o modelo Cassandra não foi preparado para receber todos os dados do PostgreSQL, apenas os importantes para esta trilha de auditoria. A eficiência nesta consulta foi de 47,36%.

O Cassandra se mostrou eficaz para solucionar o problema proposto nesta monografia. A forma de particionar os dados e o uso dos *snitches* traz vantagens pelo fato de distribuir os dados de forma uniforme entre os nós do Cassandra, deixando-os bem balanceados. Além disso, poder utilizar dados de vários formatos e livres de esquemas, traz uma grande flexibilidade tornando o uso do Cassandra possível para diversos estudos de casos.

O Cassandra tem a desvantagem de ser modelado baseado nas consultas dos dados, para projetar o banco é necessário saber exatamente a forma como os dados serão utilizados. Diferente dos bancos de dados relacionais, que persistem os dados de forma a ser possível fazer consultas não planejadas na modelagem.

Como o Cassandra foi construído para manter dados em ambientes computacionais com vários nós, onde cada um pode estar localizado fisicamente em locais diferentes, um trabalho futuro poderia fazer análise considerando quantidades diferentes de nós para avaliar performance. É possível, também, fazer um trabalho futuro que faça comparações

de inserção e recuperação de dados, além da consulta da trilha de auditoria de incompatibilidade de rubrica, utilizando outros modelos NoSQL.

Referências

[1] *Apache Cassandra^{TM} 2.0 Documentation.* vii, 13, 16, 17, 18, 19, 20, 21, 22, 23, 37

[2] Considerações sobre o banco de dados apache cassandra. `http://www.ibm.com/developerworks/br/library/os-apache-cassandra/`. Acessado em 23/08/2014. vii, 15

[3] Cql for cassandra 2.0 documentation. `http://www.datastax.com/documentation/cql/3.1/pdf/cql31.pdf`. Acessado em 28/08/2014. 16

[4] Evaluate and learn pentaho data integration. `https://help.pentaho.com/Documentation/5.1/0D0/1A0/010/000`. Acessado em 28/08/2014. 28

[5] Hector a high level java client for apache cassandra. `http://hector-client.github.io/hector/build/html/index.html`. Acessado em 20/08/2014. 29

[6] Siape - sistema integrado de administração de recursos humanos. `https://www.serpro.gov.br/conteudo-solucoes/produtos/administracao-federal/siape-sistema-integrado-de-administracao-de-recursos-humanos`. Acessado em 07/08/2014. 1, 24

[7] Siapenet. `https://www.serpro.gov.br/conteudo-solucoes/produtos/administracao-federal/siapenet`. Acessado em 07/08/2014. 24

[8] What is apache cassandra? `http://planetcassandra.org/what-is-apache-cassandra/`. Acessado em 03/08/2014. 2, 13

[9] What is nosql? `http://planetcassandra.org/what-is-nosql/`. Acessado em 16/08/2014. 1

[10] Daniel Abadi, Peter Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009. 34

[11] Ramez Elmasri, Shamkant Navathe, Marília Guimarães Pinheiro, Cláudio César Canhette, Glenda Cristina Valim Melo, Claudia Vicei Amadeu, and Rinaldo Macedo de Morais. *Sistemas de banco de dados.* Pearson Addison Wesley, 2005. 4, 5, 7, 8

[12] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011. 9

[13] Carlos Alberto Heuser. *Projeto de banco de dados.* Série Livros Didáticos. Sagra Luzzatto, 1998. vii, 4, 5, 6, 7

[14] Eben Hewitt. *Cassandra: The Definitive Guide.* Definitive Guide Series. O'Reilly Media, 2010. 14, 15, 16

[15] Ruben Huacarpuma, Daniel Rodrigues, Antonio Rubio Serrano, João Paulo Lustosa da Costa, Rafael de Sousa Júnior, Maristela Holanda, and Aleteia Araujo. Big data: A case study on data from the brazilian ministry of planning, budgeting and management. vii, 24, 25, 32, 36

[16] Raghu Ramakrishnan and Johannes Gehrke. *Sistemas de gerenciamento de banco de dados - 3.ed.* McGraw Hill Brasil, 2008. 5

[17] Eric Redmond, Jim Wilson, and Jacqueline Carter. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement.* Oreilly and Associate Series. Pragmatic Bookshelf, 2012. 9, 10, 11

[18] Shashank Tiwari. *Professional NoSQL.* Programmmer to programmer. Wiley, 2011. 5, 8, 9, 11, 12