UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO

ALBERTO MARTINEZ SCREMIN
BIANCA CARUSO DA PAIXÃO

ANALISE DE BANCO DE DADOS NÃO RELACIONAIS E COMPARAÇÃO COM BANCO DE DADOS RELACIONAIS

NITERÓI
2011

ALBERTO MARTINEZ SCREMIN
BIANCA CARUSO DA PAIXÃO

UM ESTUDO SOBRE BANCOS DE DADOS NÃO RELACIONAIS

Trabalho de Conclusão de Curso Apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: VANESSA BRAGANHOLO MURTA

NITERÓI
2011

ALBERTO MARTINEZ SCREMIN
BIANCA CARUSO DA PAIXÃO

ANALISE DE BANCO DE DADOS NÃO RELACIONAIS E COMPARAÇÃO COM BANCO DE DADOS RELACIONAIS

Trabalho de Conclusão de Curso Apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense para obtenção do grau de Bacharel em Ciência da Computação.

Data da Aprovação: _____

BANCA EXAMINADORA

_____
Prof. VANESSA BRAGANHOLO MURTA Orientador
UFF

_____
Prof. Alexandre Plastino de Carvalho
UFF

_____
Prof. Luiz André Portes Paes Leme
UFF

NITERÓI 2011

Dedicamos este trabalho aos nossos pais que nos apoiaram durante todo o caminho traçado durante nossa vida acadêmica. Agradecemos também a professora Vanessa que nos orientou durante este trabalho.

AGRADECIMENTOS

Agradecemos, primeiramente, aos nossos pais que nos apoiaram durante todos os anos e em todas as fases. Devido a eles, nós tivemos condição de estudar e chegar até esta etapa de nossas vidas. Além disso, gostaríamos de agradecer aos nossos irmãos e familiares pelo carinho e apoio.

À nossa professora e orientadora, Vanessa Braganholo, pela paciência na orientação e incentivo que tornaram possível a conclusão desta monografia.

Aos funcionários da coordenação e da secretaria da UFF que nos acompanharam nesta jornada e estavam sempre dispostos a ajudar, em qualquer horário.

Aos nossos colegas de classe, que compartilharam diversos momentos, bons e ruins durante todos esses anos de atividade acadêmica.

RESUMO

Os SGBDs relacionais têm sido extremamente utilizados em praticamente todos os sistemas desenvolvidos nos últimos tempos. Estes oferecem muitas vantagens aos seus usuários, e possuem diversos mecanismos que permitem o controle da concorrência, segurança, integridade dos dados entre outros. Entretanto, nos últimos anos, os bancos de dados não relacionais, também conhecidos como NoSQL, ganharam grande força no mercado. O crescimento do volume dos dados das organizações, além de outros fatores limitantes, tais como a estrutura pouco flexível dos modelos relacionais e a questão da escalabilidade dos sistemas, foram os responsáveis pelo ganho de força do NoSQL. Tendo em vista este cenário, este trabalho faz uma análise de algumas características de SGBDs relacionais e não relacionais, apresentando uma comparação de aspectos ligados à modelagem e consulta.

Palavras-chaves: Bancos de Dados, NoSQL, Modelo Relacional, Chave-Valor, XML.

ABSTRACT

Relational Database Management Systems have been largely used in a lot of systems developed in recent times. They offer many advantages to theirs users, including several mechanisms that allow concurrency control, security, data integrity among others. However, in recent years, non-relational databases, also known as NoSQL, have grown strong in the market. The volume of data in organizations, and other limiting factors such as the inflexibility of the structure of the relational model, and also issues about system scalability, was responsible for this rise. In this contexto, this work examines some characteristics of relational and non-relational databases, and presents a comparison between those models in terms of data model and quering.

Keywords: Databases, NoSQL, Relational Databases, key-value, XML

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ANSI: *American National Standards Institute*

BI: Business Intelligence

CLI: Command Line Interface

CSV: *Comma-Separated Values*

IP: Internet Protocol

ISO: *International Standards Organization*

NoSQL: *Not Only SQL*

OLTP: Online Transaction Processing

SGBD: Sistema de Gerenciamento de Banco de Dados

SQL : *Structured Query Language*

SSL: Secure Sockets Layer

TB: Terabyte

XML: Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Atualmente, a maioria das grandes aplicações *web* encaram um grande desafio: servir milhões de usuários distribuídos por todo o mundo, os quais esperam que o serviço esteja disponível a todo tempo. As grandes aplicações de sucesso não possuem apenas uma grande base de usuários. Elas também estão crescendo mais rapidamente do que o desempenho do *hardware* (STEPPAT, 2009). Sendo assim, tais aplicações precisam possuir a habilidade de escalar.

A necessidade de escalar as aplicações varia dependendo dos objetivos de cada uma. Algumas precisam da habilidade de servir um grande número de usuários ao mesmo tempo, outras possuem a necessidade de armazenar de maneira distribuída grandes volumes de dados. Algumas aplicações com caráter social, como as conhecidas redes de relacionamento, podem precisar escalar com ambos objetivos.

O *NoSQL* é um movimento que surgiu no início de 2009, com a intenção original de desenvolver bancos de dados modernos, voltados para a *web* e escaláveis. O termo *NoSQL* é a abreviação de *Not Only SQL*. Essa terminologia designada pela comunidade sugere a idéia de que o *SQL* não é necessariamente a única opção que existe para o desenvolvimento de aplicações, principalmente aplicações *web* (POPESCU, 2009).

A grande motivação para o movimento *NoSQL* foi resolver o problema de escalabildade dos bancos tradicionais. Pode ser muito caro e complexo escalar um banco de dados relacional (cuja linguagem de consulta é o SQL). Esse movimento está bastante enraizado no *open source*. É possível perceber isto através dos nomes curiosos dos projetos: Voldemort, MongoDB, Cassandra, Tokyo Tyrant e CouchDB (BOGONI, 2010).

O movimento NoSQL ganhou muita força quando algumas famosas redes sociais começaram a adotar esta tecnologia, como foi o caso do *Twitter*. O microblog trata da escrita e da leitura de grandes quantidades de dados, cerca de 7TB por dia. Este pode ser um desafio para os bancos de dados relacionais. Entretanto, a adoção do paradigma *NoSQL* pela equipe de desenvolvimento do Twitter permitiu a distribuição dos dados entre múltiplos nós. Esta habilidade levou à migração do MySQL para o Cassandra (BOGONI, 2010).

É importante lembrar que apesar de serem projetados para suportar desempenho e escalabilidade, os bancos de dados *NoSQL* não suportam as propriedades Atomicidade, Consistência, Isolamento e Durabilidade (ACID) (ELMASRI; NAVATHE, 2009), que são atendidas pelos bancos de dados relacionais. Os desenvolvedores acostumados com o padrão SQL utilizado nos bancos de dados relacionais, não encontrarão o mesmo padrão se usarem implementações *NoSQL*, pois cada implementação usa seu próprio mecanismo de acesso.

1.2 OBJETIVO

Este trabalho propõe mostrar algumas características dos bancos de dados relacionais e não relacionais nos quesitos modelo de dados, consulta, indexação e otimização de consultas. Através desta análise, pretendemos facilitar o entendimento de como funciona esta tecnologia que está

ganhando muita força – o NoSQL – e além disso, mostrar quais os seus aspectos que devem ser explorados.

Cada tipo de banco de dados possui características marcantes que facilitam o desenvolvimento e aumentam o desempenho das aplicações. Queremos mostrar quais são estes pontos facilitadores e incentivar o uso da tecnologia mais adequada de acordo com a necessidade das aplicações, evitando assim algumas limitações enfrentadas pelos desenvolvedores.

Para atingir esse objetivo, foram escolhidos um SGBD de cada tipo. O MySQL, um banco de dados relacional; o Sedna, um banco de dados XML; o Cassandra e o Redis, bancos de dados chave-valor e o Mongodb, um banco de dados orientado a documentos. Desta forma, podemos compará-los e entender as diferenças entre as tecnologias.

Em um segundo momento, utilizamos uma ferramenta de benchmark para gerar um modelo de uma aplicação no esquema relacional. Em seguida, mostramos como este modelo pode ser representado em cada um dos SGBDs selecionados e como as consultas são realizadas.

1.3 ORGANIZAÇÃO DO TEXTO

O Capítulo 2 descreve cada um dos SGBDs estudados: MySQL, Sedna, Cassandra, Redis e Mongodb. Todos são descritos de acordo com os critérios modelo de dados, linguagem de consulta utilizada e índices. Também são fornecidos alguns exemplos de consultas e o trabalho do otimizador de consultas é ilustrado.

O Capítulo 3 exemplifica um esquema relacional fornecido por um benchmark, e como este esquema seria representado nos bancos de dados não relacionais que estudados neste trabalho. Além disso, o capítulo mostra uma comparação das formas de consulta em cada um dos bancos de dados estudados.

Finalmente, o Capítulo 4 mostra as conclusões que foram tiradas deste estudo e também algumas propostas para trabalhos futuros.

2 BANCOS DE DADOS RELACIONAIS E NoSQL

Neste capítulo são descritos todos os *SGBDs* relacionais e não relacionais estudados. Para cada *SGBD* são levados em consideração alguns critérios importantes para a comparação entre eles. O modelo de dados, a linguagem de consulta, os mecanismos de indexação e de otimização das consultas são os pontos descritos em cada *SGBD*.

2.1 MYSQL

O MySQL é um sistema de gerenciamento de banco de dados, que utiliza a linguagem SQL (BEAULIEU, 2009) como interface. O MySQL possui algumas características que o definem, como portabilidade, pois funciona em praticamente qualquer plataforma atual, compatibilidade com diversos drivers e linguagens de programação, é *open source*, entre outras características (DUBOIS, 2000). Além disso, o MySQL também é reconhecido pelo seu desempenho, robustez e por possuir diversos mecanismos como *backup*, *restore* e replicação. Tais mecanismos facilitam seu uso em ambientes que requerem alta disponibilidade do banco de dados (DUBOIS, 2008).

2.1.1 Modelo de Dados

O MySQL é um banco de dados relacional (CODD, 1970), cujo modelo é composto por tabelas que mantêm linhas e colunas. Cada linha, também conhecida como tupla, contém dados relacionados sobre uma determinada entidade, como por exemplo, a entidade pessoa. Cada coluna representa um pedaço de dados sobre esta entidade, como nome, endereço, telefone (STEPHENS, 2009).

Cada entrada em uma determinada coluna de uma tabela deverá conter o mesmo tipo de dados. O conjunto dos valores que são permitidos para uma coluna é chamado de domínio da coluna. O domínio está intimamente relacionado ao tipo de dados, mas não são a mesma coisa. Um tipo de dados da coluna é o tipo de dados que a coluna pode conter. Os tipos de dados de uma coluna dependem do banco de dados específico que está sendo utilizado (STEPHENS, 2009).

No MySQL encontramos 3 tipos de dados bem comuns: inteiros, data e hora, e texto. Os tipos inteiros podem ser *bit, smallint, bigint, int, float, double*, entre outros. Os tipos textos podem ser *char, varchar, binary, text, blob,* e os campos de data podem ser principalmente *datetime, date, time, year* (DUBOIS, 2008).

A Figura 1 mostra um exemplo da estrutura de uma tabela que representa uma entidade pessoa. Os atributos da tabela Pessoa são nome, CPF e data de nascimento. Além disso, esta tabela também possui uma chave primária na coluna CPF que é o identificador único da tabela. As chaves das tabelas são explicadas na seção 2.1.3.

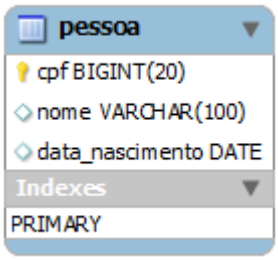

Figura 1: Exemplo de Tabela no Modelo Relacional.

As instâncias da tabela pessoa são representadas por suas tuplas. No exemplo da Figura 2 existem duas tuplas: <08539287409, João, 17-05-1967> e <05831765208, Maria, 21-10-1990>.

Pessoa

cpf	nome	data_nascimento
08539287409	João	17-05-1967
05831765208	Maria	21-10-1990

Figura 2: Representação das tuplas no modelo relacional.

Além das tabelas, colunas e índices, os bancos de dados relacionais oferecem mais alguns recursos que facilitam a sua modelagem e o manuseio, como as chaves estrangeiras, *triggers*, *stored procedures* e as funções.

As chaves estrangeiras, no MySQL, mantêm a integridade referencial das tabelas, ou seja, são utilizadas para definir as relações entre duas ou mais tabelas normalmente no contexto "tabela pai - tabela filha". Neste caso não é possível criar um registro em uma coluna que referencie um dado inexistente na tabela pai. Este tipo de tratamento facilita o desenvolvimento, pois evita que essas regras tenham que ser implementadas na aplicação (SCHNEIDER, 2005).

As *stored procedures* são trechos de código armazenados no banco que acessam as tabelas e realizam operações sobre os dados. Estes procedimentos podem receber parâmetros e também podem retornar resultados. As funções possuem uma finalidade semelhante às *stored procedures*, entretanto com ligeiras diferenças. As funções necessariamente retornam um resultado e, além disso, não conseguem acessar as tabelas do banco (SCHNEIDER, 2005).

Os gatilhos (*triggers)* introduzem um tratamento de eventos no lado do servidor e controle para o banco de dados. Um gatilho pode ser configurado para atuar antes ou após qualquer evento de modificação de dados das tabelas e realizar outras operações, tanto com os dados novos, como com os dados antigos (SCHNEIDER, 2005).

Os bancos de dados relacionais funcionam bem de acordo com as necessidades específicas do sistema, além de prover uma grande flexibilidade na criação de novas consultas, sem que estas

estivessem planejadas no início da modelagem do projeto. Além disso, o banco de dados relacional também funciona bem se a aplicação necessitar executar consultas complexas com muitas junções e necessitar da integridade referencial como parte das regras de negócio.

Entretanto, existem casos em que o banco de dados relacional não tem bom desempenho. Suponha que uma aplicação precise armazenar arquivos binários, ou então que os dados sejam estruturados como uma hierarquia ou um grafo. Neste caso, é necessário utilizar uma topologia especial para realizar as principais funções da aplicação. Além disso, os bancos relacionais trazem mais complicações caso o sistema possua um modelo de dados que sofra alterações constantemente (STEPHENS, 2009).

2.1.2 Linguagem de Consulta

O MySQL utiliza uma linguagem de consulta muito conhecida pela maioria dos administradores de banco de dados: o SQL (BEAULIEU, 2009). SQL é uma linguagem bastante simples, porém com grande poder de expressão.

O SQL possui algumas subdivisões:

- DML (Linguagem de Manipulação de Dados) – é utilizado para realizar inclusões, exclusões e alterações de dados presentes em registros. Os principais comandos são: *INSERT*, *UPDATE* e *DELETE.*

- DDL (Linguagem de Definição de Dados) – é utilizado para definir tabelas novas e elementos associados, ou para alterar uma tabela, adicionando uma nova coluna. Os principais comandos são: *CREATE, DROP, ALTER TABLE.*

- DCL (Linguagem de Controle de Dados) – é utilizado para controlar os aspectos de autorização de dados e licenças de usuários visando à segurança da aplicação. Os principais comandos são: *GRANT, REVOKE.*

- DTL (Linguagem de Transação de Dados) – é usado para marcar o começo de uma transação de banco de dados que pode ser completada ou não, por exemplo: *START TRANSACTION.* Além disso, existe também o comando *COMMIT* que realiza todas as mudanças dos dados permanentemente, e o *ROLLBACK,* que faz com que as mudanças nos dados existentes desde o último *COMMIT* ou *ROLLBACK* sejam descartadas.

- DQL (Linguagem de Consulta de Dados) – é utilizada para realizar uma consulta na base de dados de forma a encontrar um resultado específico. O único comando desta subdivisão é o *SELECT*.

O SELECT é um dos comandos SQL mais utilizados. Este comando é composto por várias cláusulas e operadores, possibilitando elaborar consultas das mais simples às mais elaboradas. As cláusulas são modificadores utilizados para definir os dados a serem selecionados para a manipulação.

A cláusula FROM é utilizada para especificar quais tabelas serão consultadas. A cláusula WHERE é utilizada para especificar as condições que devem reunir os registros que serão selecionados. A cláusula GROUP BY é utilizada para unir registros em grupos específicos. O HAVING é utilizado para expressar a condição que cada grupo deve satisfazer. O ORDER BY é utilizado para ordenar os registros em uma ordem específica. O DISTINCT é usado para selecionar dados sem repetição.

Os operadores lógicos podem ser representados pelas cláusulas AND, OR e NOT. Além disso, também existem operadores relacionais, utilizados para realizar comparações entre valores. Os mesmos são representados pelos sinais de igual, maior que, menor que, diferente, entre outros. Por último, existem também as funções de agregação, utilizadas para devolver um único valor que se aplica a um grupo de registros. Tais funções são representadas pelos comandos AVG, COUNT, SUM, MAX e MIN.

Embora o SQL tenha sido padronizado pela ANSI e ISO (ANSI/ISO, 1986) na década de 80, a linguagem possui muitas variações e extensões produzidas pelos diferentes fabricantes de sistemas gerenciadores de bases de dados. Sendo assim, estas subdivisões e os comandos podem sofrer alterações dependendo do SGBD.

Na Figura 3 é representado o comando, no MySQL, para criar a estrutura da tabela Pessoa (Figura 1) e na Figura 4 são apresentados os comandos de inserção de dados nesta tabela. No exemplo, estamos inserindo duas linhas, a primeira referente às informações de João e a segunda referente à Maria.

```
CREATE TABLE pessoa (
        'cpf' bigint(20) unsigned NOT NULL,
        'nome' varchar(100) default NULL,
        'data_nascimento' date default NULL,
        PRIMARY KEY(`cpf`)
) ;
```

Figura 3: Comando SQL utilizado para criar tabelas no MySQL.

```
INSERT INTO pessoa (cpf, nome, data_nascimento) VALUES (08539287409, 'Joao', '1967-05-17');
INSERT INTO pessoa (cpf, nome, data_nascimento) VALUES (05831765208, 'Maria', '1990-10-21');
COMMIT;
```

Figura 4: Comandos de inserção de dados nas tabelas MySQL.

2.1.3 Índices

Os índices são os principais meios de acelerar o acesso ao conteúdo das tabelas, especialmente se a consulta envolver múltiplas junções entre as tabelas. Sem o índice, o MySQL é obrigado a ler a tabela por completo, desde a primeira linha até a última linha da tabela para responder qualquer consulta. Quanto maior for esta tabela, maior será o custo de processamento da consulta. Se a tabela possui índice sobre as colunas que são referenciadas na cláusula *WHERE*, o MySQL consegue

facilmente determinar a posição que deve procurar no arquivo de dados, sem ter que acessar todos os dados.

A maioria dos índices do MySQL são armazenados em Árvores B, como por exemplo os índices chave (*key index)*, os índices da chave primária (*Primary Key*) e da chave única (*Unique Key*). Entretanto, alguns tipos específicos de tabelas também suportam *Hash Indexes*.

A chave primária é uma superchave, ou seja, é um conjunto de colunas para as quais duas linhas diferentes não podem possuir o mesmo valor. Esta chave é utilizada como um identificador único das linhas da tabela. Uma tabela pode ter apenas uma única chave primária. Essa chave pode ser simples quando formada por apenas uma coluna, ou composta, quando formada por um conjunto de colunas (STEPHENS, 2009).

Quando especificada uma restrição de chave única para uma coluna, o MySQL gera um índice único e bloqueia qualquer tentativa de inserção ou atualização dos dados que gerem linhas duplicadas na tabela. Caso ocorra uma tentativa de duplicar os valores, um erro é gerado pelo MySQL. Desta maneira, a chave única é utilizada para identificar unicamente as linhas da tabela. A diferença desta chave para a chave primária é que as colunas que pertencem à chave primária necessariamente devem ser preenchidas, ou seja, dizemos que elas são *NOT NULL*. No caso da chave única, essas colunas podem receber valores nulos, ou seja, são colunas *NULLABLE* (SCHNEIDER, 2005).

Os índices chave são criados em qualquer coluna da tabela e não há restrições de unicidade. São utilizados para acelerar a consulta, reduzindo seu tempo de resposta.

O MySQL é bem flexível na construção dos índices. É possível indexar uma única coluna, ou uma combinação de colunas. Além disso, é possível ter mais de um índice por tabela. O número máximo de índices por tabela e o tamanho máximo de um índice varia de acordo com a forma de armazenamento que foi adotado na criação da tabela. Todas elas suportam no mínimo 16 índices por tabela. Se a coluna for do tipo *string*, é possível indexar somente os *n* primeiros caracteres da coluna, tornando o índice menor e mais rápido. No MySQL, existe uma forma de armazenamento denominada MyISAM. Nesta é possível criar *fulltext* índices que são utilizados para consultas no texto inteiro.

Em alguns casos o MySQL não utiliza o índice, mesmo se possuir algum disponível. Um caso em que isso ocorre é quando o otimizador estima que a utilização do índice irá requerer muitos acessos ao disco. Isso significaria em um grande percentual de linhas da tabela sendo lido. Neste caso, ler a tabela por completo é mais rápido. Entretanto, se a mesma consulta utilizar o *LIMIT*, comando que limita o tamanho do resultado em um subconjunto de linhas, o MySQL utiliza o índice.

Um exemplo de criação de índice no MySQL pode ser visto a seguir. Neste comando é criado um índice chamado id_nome sobre a coluna nome da tabela pessoa.

CREATE INDEX id_nome ON pessoa(nome);

O MySQL possui um otimizador que provê informações do plano de execução das consultas. Além disso, descreve como realizar a configuração dos *caches* das chaves para armazenar informações dos índices em memória.

Quando uma consulta não executa tão rápido quanto se gostaria, é possível utilizar a cláusula *EXPLAIN* para solicitar ao servidor MySQL informações sobre como o otimizador de consulta processa a consulta. Esta informação pode ser útil por diversos motivos:

- *EXPLAIN* pode informar pontos da consulta que precisam de índices;
- Se a tabela já possui um índice, o *EXPLAIN* mostra se o otimizador está utilizando o mesmo;
- Se o índice existe, mas não está sendo utilizado, o analista pode reescrever a consulta de diversas maneiras diferentes. O *EXPLAIN* pode indicar qual das consultas é a melhor para ajudar o servidor a utilizar os índices disponíveis.

Para utilizar este mecanismo de otimização, basta escrever a consulta normalmente e colocar a cláusula *EXPLAIN* no início da consulta. A seguir é mostrado um exemplo de duas consultas feitas na tabela Pessoa, descrita na Figura 1. Assumindo que foram inseridos 670 registros na base de dados, e que existe apenas um João, cujo CPF é 08539287409, as duas consultas retornam o mesmo resultado. Porém, elas não são igualmente eficientes.

```
mysql> explain select * from pessoa where nome = 'Joao';
*************************** 1. row ***************************
           id: 1
select_type: SIMPLE
        table: pessoa
         type: ALL
possible_keys: NULL
          key: NULL
key_len: NULL
          ref: NULL
         rows: 506
        Extra: Using where
1 row in set (0.00 sec)

mysql> explain select * from pessoa where cpf = 08539287409;
*************************** 1. row ***************************
           id: 1
select_type: SIMPLE
        table: pessoa
         type: const
possible_keys: PRIMARY
          key: PRIMARY
key_len: 8
          ref: const
         rows: 1
        Extra:
1 row in set (0.00 sec)
```

O comando *explain* retorna diversas informações. Na primeira consulta, a palavra *NULL* nas colunas *possible_keys* e *key* mostram que nenhum índice foi considerado disponível e também nenhum foi utilizado para processar a consulta. Já na segunda consulta, o índice da chave primária da tabela é

proposto como um possível índice a ser utilizado, e é exatamente o índice que o otimizador irá escolher para trazer os resultados.

A coluna *rows* indica os efeitos desta diferença, ou seja, o valor indica a quantidade de linhas que o MySQL estima que serão analisadas enquanto processa a consulta. A primeira consulta retorna o valor 506, que é grande parte da tabela Pessoa, ou seja, o MySQL irá processar quase todas as linhas da tabela antes de encontrar o registro de João, mostrando que a consulta é extremamente ineficiente. Já na segunda consulta, somente uma linha é processada, visto que o MySQL utiliza o índice da chave primária, conseguindo ir direto para a única linha realmente relevante da consulta (DUBOIS, 2008).

2.2 SEDNA

Sedna é um banco de dados nativo XML (BRAY et al., 2008), open source, que implementa alguns serviços comuns aos bancos de dados relacionais, como: armazenamento persistente, transações ACID, segurança, índices. Neste SGBD, as consultas são realizadas em XQuery (BOAG et al., 2010), ou através do XPath (CLARK; DEROSE, 2003).

Sedna possui como objetivo prover um ambiente em tempo de execução para aplicações que processam dados XML. Isso implica uma estreita integração das funcionalidades da gestão de banco de dados com a linguagem de programação (CUONG, 2006).

2.2.1 Modelo de Dados

No Sedna, o modelo de dados utilizado é o XML. XML é uma linguagem poderosa para a representação de dados semi-estruturados. Dados semi-estruturados são aqueles que contêm seus esquemas embutidos dentro dos dados. Dessa forma, se auto-descrevem (BUNEMAN, 1997).

XML é simples, flexível e derivado do SGML. Foi projetado para superar o desafio da publicação de arquivos em larga escala, mas atualmente tem um papel grande na troca de informações na Internet (BRAY et al., 2008). Um documento XML tem que seguir algumas especificações para que seja bem formado. São elas:

- Devem possuir uma raiz única;
- Todos os elementos possuem uma tag inicial e uma tag final;
- Elementos têm que ser bem aninhados, ou seja, se um elemento A contém como filho um elemento B, a tag final de B aparece antes da tag final de A;
- Atributos não se repetem dentro de um mesmo elemento;
- Elementos são sensíveis a maiúsculas e minúsculas (BRAY et al., 2008).

Na Figura 5 temos o exemplo de um arquivo XML bem formado. No exemplo, o elemento raiz é *pessoas*. Ele contém todos os outros elementos do documento.

```
<pessoas>
  <pessoa>
    <cpf>08539287409</cpf>
    <nome>Joao</nome>
    <data_nascimento>1967-05-17</data_nascimento>
  </pessoa>
  <pessoa>
    <cpf>05831765208</cpf>
    <nome>Maria</nome>
    <data_nascimento>1990-10-21</data_nascimento>
  </pessoa>
</pessoas>
```

Figura 5: Exemplo de Documento XML

2.2.2 Linguagem de Consulta

XQuery é uma linguagem de consultas padrão da W3C (BOAG et al., 2010) utilizada pelo Sedna para a manutenção de seus dados. A XQuery permite consultas a todos elementos de documentos XML.

Um exemplo de consulta para que sejam retornados todos os nomes das pessoas do documento da Figura 5 pode ser expressa, em XPath, da seguinte forma. Nesta consulta, estamos selecionando o elemento nome, que é filho do elemento pessoa, no documento XML "pessoas.xml". Note que o elemento pessoa não é a raiz do documento. Neste caso, é necessário usar o operador //, que serve para recuperar um elemento pessoa em qualquer nível de profundidade na árvore XML.

doc("pessoas.xml")//pessoa/nome

A mesma consulta também poderia ser expressa como uma consulta XQuery. A mesma é representada na Figura 6.

```
for $p in doc("pessoas.xml")//pessoa
return
<pessoa>
  {$p/nome>
</pessoa>
```

Figura 6: Exemplo de XQuery para buscar o nome de uma pessoa.

2.2.3 Índices

Sedna utiliza árvores B para armazenar seus índices. Os índices permitem que um elemento seja indexado pelo valor de um de seus subelementos. Como um exemplo, na Figura 7 o CPF está sendo usando para indexar os elementos pessoa do documento XML da Figura 5.

O primeiro caminho descrito na Figura 7 (doc("pessoas")//pessoas/pessoa), expresso por uma expressão XPath, indica qual elemento vai ser retornado por esse índice. O segundo caminho (cpf),

também expresso por uma expressão XPath (relativa à primeira), indica o campo que vai ser indexado. Nota-se que o tipo desse campo deve ser informado explicitamente.

```
CREATE INDEX “cpf” ON doc(“pessoas”)/pessoas/pessoa BY cpf AS xs:string
```

Figura 7: Criação de índice no Sedna

Uma das grandes vantagens de um SGBD XML é o fato da indexação ser *full text*. O XML Schema associado tem a função de servir como ponto de entrada para a estrutura do documento XML. Assim, as consultas executadas sobre esse documento acessam os elementos como se fossem nós, fazendo com que o esquema descritivo possa ser utilizado como um índice natural para avaliação de expressões XPath (TARANOV et al., 2010).

O Sedna apresenta algumas camadas para otimização de consultas baseadas em regras. Ele utiliza-se de seis regras para otimização de consultas (GRINEV et al., 2004):

- Técnica de aninhamento de funções: troca as chamadas das funções pelas funções em si, ou seja, as expandem, tornando as outras técnicas de otimização mais fáceis de serem implementadas;
- Adiamento da construção de elementos XML em predicados: através desta técnica, a tarefa de construção de elementos XML, que tem alto custo, é feita apenas após o resultado já ter sido filtrado;
- Simplificação de consultas através de esquema associado: permite que o usuário que não conheça o esquema associado ao documento que está sendo consultado consiga um melhor desempenho em suas consultas. Essa técnica é baseada na inferência estática da XQuery, criando consultas mais precisas;
- Fazer com que consultas sejam as mais declarativas possíveis: permite que o otimizador possa procurar por um conjunto maior de planos de execução das consultas;
- Normalização de junções: consiste em reescrever as consultas de forma que possa ser aplicada não apenas iterações aninhadas, mas também outros algoritmos de junções. O Sedna consegue isso através da extração de expressões XPath de dentro das junções;
- Identificação de operações livres de iteração dentro do corpo de iterações: isso reduz o custo das consultas retirando estes tipos de ocorrência, reduzindo assim o número de execuções desnecessárias de certas operações.

2.3 CASSANDRA

O Cassandra é um SGBD *open source* não relacional. Apesar de muitos o denominarem como um banco de dados orientado a colunas, ele pode ser encarado como um índice. Isso porque o Cassandra utiliza o armazenamento dos dados baseado em linhas, onde cada linha possui uma chave única que torna o dado acessível (HEWITT, 2011). Sabendo disso, podemos classificar o Cassandra como um SGBD orientado a Chave-Valor (HEWITT, 2011).

2.3.1 Modelo de Dados

O modelo de dados do Cassandra funciona ligeiramente diferente dos modelos relacionais. Foram criados alguns conceitos completamente novos, como o *keyspace*. Entretanto, também existem conceitos que pertencem ao modelo relacional e ao modelo chave-valor, porém com significados diferentes, como é o caso das colunas.

O conceito do *keyspace* é semelhante ao do banco de dados no modelo relacional. No modelo relacional, o banco de dados é o repositório das tabelas e assim como ele, o *keyspace* é considerado o recipiente de pelo menos uma família de colunas. Cada *keyspace* possui um nome e uma lista de atributos que definem o seu comportamento.

Uma família de colunas (*column family*) é análoga às tabelas no modelo relacional. Ela representa a estrutura dos dados e é considerada o repositório para as coleções de linhas. Estas linhas são formadas por um conjunto de colunas, mas este conjunto não é necessariamente igual para todas as linhas (HEWITT, 2011).

O modelo da Figura 8 mostra um exemplo da família de colunas Pessoa, composta por duas linhas, onde a chave da linha é o CPF. Cada linha possui duas colunas, que são os atributos nome e data de nascimento.

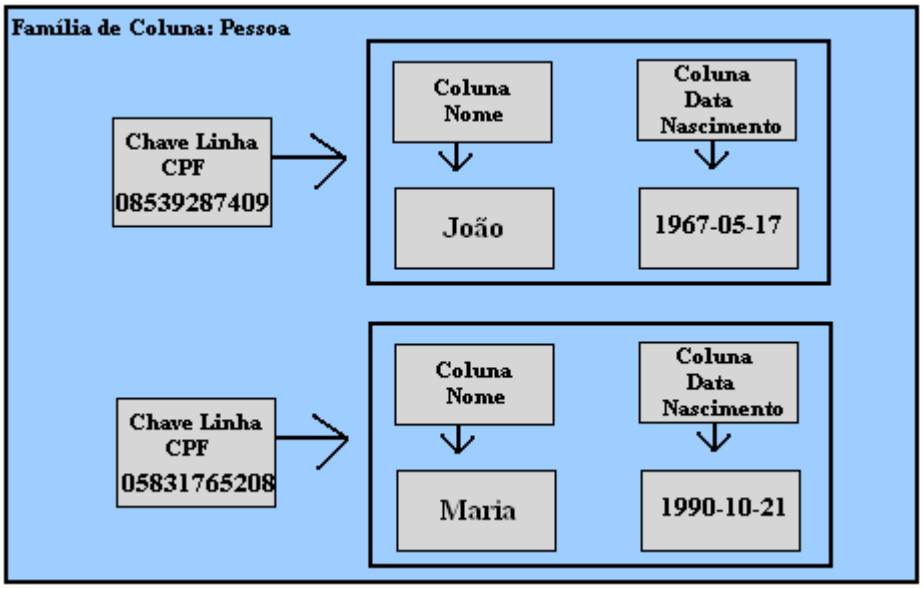

Figura 8: Modelo de dados do Cassandra.

Podemos representar os dados deste modelo (Família Pessoa) como é mostrado abaixo:

Pessoa:

08539287409:

Nome: Joao

DataNascimento: 1967-05-17

05831765208:

Nome: Maria

DataNascimento: 1990-10-21

2.3.2 Linguagem de Consulta

Existem algumas diferenças entre as consultas realizadas no Cassandra e nos SGBDs relacionais, que são explicadas a seguir (HEWITT, 2011).

- O Cassandra não realiza atualizações de registros, ou seja, não existe um comando de *update*. Para atualizar um valor, basta inseri-lo em uma chave já existente. Sempre que isso ocorre, o Cassandra substituiu o dado existente pelo dado que foi inserido.
- O Cassandra não possui suporte a transações.
- Ele não permite chaves duplicadas, ou seja, se o usuário tentar inserir dados relativos a uma chave que já existe no banco, os dados cadastrados sob aquela chave são sobrescritos.
- As escritas no Cassandra são muito rápidas, pois foi implementado para não realizar idas ao disco durante este processo.

Nos exemplos das Figura 9 e Figura 10 é possível verificar exemplos de criação da *keyspace* e da família de colunas, respectivamente. Na Figura 11, Figura 12 e Figura 13, são apresentados os comandos de inserção de dados, consultas e remoção de dados, respectivamente. Todos os comandos foram executados no *CLI*, um *client* do Cassandra.

```
create keyspace usuarios;
```

Figura 9: Criando uma keyspace.

```
create column family pessoa with comparator=UTF8Type
and column_metadata=[{column_name: nome, validation_class: UTF8Type},
{column_name: data_nascimento, validation_class: LongType}];
```

Figura 10: Consulta utilizada para criar a estrutura da família de colunas Pessoa.

```
set pessoa['08539287409']['nome'] = 'Joao';
set pessoa['08539287409'][ 'data_nascimento'] = '1967-05-17';
set pessoa['05831765208']['nome'] = 'Maria';
set pessoa['05831765208']['data_nascimento'] = '1990-10-21';
```

Figura 11: Comandos de inserção no Cassandra.

```
get pessoa where nome = 'Joao';
get pessoa where data_nascimento = '1967-05-17';
```

Figura 12: Comandos de consulta no Cassandra.

```
del pessoa['08539287409']['nome'];
del pessoa['08539287409'];
```

Figura 13: Comandos de remoção de uma coluna e de uma linha da família.

No Cassandra, podemos criar uma coluna com um tempo de expiração, ou seja, dado este tempo, a coluna é apagada. Para isto, basta colocar no comando de *set* a seguinte cláusula: *withttl*=<segundos>;

2.3.3 Índices

O Cassandra cria um índice para a chave primária automaticamente. Este índice é criado para todas as famílias de coluna. A última versão lançada do Cassandra (versão 0.7) passou a suportar índices secundários (índices nas colunas) que são índices *hash* e permitem consultas por qualquer valor. Além disso, podem ser construídos em segundo plano automaticamente, sem bloquear leituras e escritas. Entretanto, esse tipo de índice possui uma série de limitações. São elas (ELLIS, 2010):

- Por ser um índice *Hash*, não permite a consulta por um subconjunto de valores. O índice só é utilizado em consultas de igualdade.
- É recomendado utilizar índices em atributos que possuem baixa cardinalidade.

Para adicionar um índice na coluna nome, deve-se utilizar o comando a seguir.

```
update column family pessoa with comparator=UTF8Type
and column_metadata=[ {column_name: nome, validation_class: UTF8Type, index_type: KEYS
},{column_name: data_nascimento, validation_class: LongType }];
```

O Cassandra utiliza as estatísticas para a escolha dos índices que serão usados na consulta, de forma que lê a menor quantidade de linhas possíveis.

2.4 MONGODB

MongoDB é um banco de dados que faz a utilização de coleções ao invés de tabelas como nos bancos relacionais e da utilização de documentos que podem ser vistos como as linhas dos bancos relacionais. Dessa forma, ele é classificado como um Banco Orientado a Documentos.

2.4.1 Modelo de Dados

O MongoDB é um SGBD orientado a documentos, ou seja, um documento é a unidade básica para representar dados no MongoDB. Cada documento no Mongo é o equivalente a uma linha em um SGBD relacional. O documento é o coração da estrutura do MongoDB. Cada documento é formado por um conjunto de chaves que são associadas a um valor (CHODROW; DIROLF, 2010). Na Figura 14, pode ser visto um exemplo de um documento no MongoDB contendo as informações de uma pessoa. Este é composto pelas chaves cpf, nome e data de nascimento que estão associadas a um valor.

Além disso, no MongoDB existe o conceito de coleções, as quais são formadas por um conjunto de documentos. Da mesma forma, uma coleção no MongoDB pode ser comparada à uma tabela no modelo relacional, porém sem nenhuma rigidez na estrutura (CHODROW; DIROLF, 2010). Além de agrupar documentos por coleções, o MongoDB também agrupa coleções por banco de dados. Uma única instância do MongoDB pode hospedar um conjunto de bases de dados totalmente independentes. Cada base de dados possui suas próprias permissões e cada banco de dados é armazenado em arquivos distintos no disco (CHODROW; DIROLF, 2010).

A idéia básica do MongoDB é substituir o conceito de “linha” do modelo relacional em um conceito mais flexível, o “documento”. O MongoDB é dito um SGBD sem um esquema definido (*schema-free*) pois não exige uma estrutura rígida para todos os documentos da coleção. Desta maneira, novos campos podem ser inseridos conforme a necessidade e os documentos de uma mesma coleção podem possuir diversos formatos (CHODROW; DIROLF, 2010).

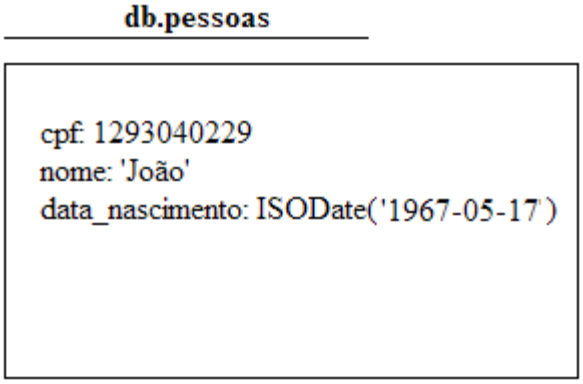

Figura 14: Documento da coleção de pessoas.

Considerando o exemplo da coleção pessoa, o MongoDB não precisa de nenhum comando para criá-la, por ser um banco livre de esquema. Uma coleção é criada a partir da inserção do primeiro documento. Para criar o documento mostrado na Figura 14 foi utilizado o comando abaixo.

db.pessoa.insert({cpf:1293040229,nome: 'João', data_nascimento: ISODate('1967-05-17')});

Este comando insere na coleção pessoa um documento formado pelas chaves cpf, nome e data_nascimento, contendo os valores 1293040229, João e 17/05/1967, respectivamente. Na seção 2.4.2 são explicados com mais detalhes os comandos de inserção, consulta, atualização e remoção.

No MongoDB não existem junções entre as coleções, que seriam as tabelas de um banco de dados relacional. Naturalmente, não se deve desnormalizar a maior parte do banco, pois não se pode contar com as junções. No lugar, o que deve ser utilizado são os documentos que contêm documentos chamados de *embedded documents*, levando assim a não termos uma coleção para cada objeto da aplicação. Normalmente, no modelo relacional, cada objeto é representado por uma tabela separada. No MongoDB isto não é recomendado, pois a utilização dos *embedded documents* é muito mais eficiente, já que os dados são armazenados em um único arquivo no disco, evitando o acesso a mais de um arquivo em disco. Diz-se que a questão principal do MongoDB consiste em escolher os objetos que devem virar coleções. Deve-se levar em conta algumas das considerações gerais, como:

- Objetos de primeira classe devem tornar-se coleções;
- Relacionamentos muitos para muitos geralmente se relacionarão por referência;
- Coleções com poucos objetos podem existir como coleções separadas;
- Documentos têm limite de 8MB, o que pode ser um limite para documentos conterem outros documentos.

Os documentos também podem ser referenciados, para fins de normalização, e contar com alguma integridade. Essa referência pode ser feita de duas formas: manualmente ou usando *dbref*. Para os casos em que a referência é feita manualmente, o desenvolvedor precisa tratar essa referência com uma consulta a mais no banco. Por exemplo, supondo o esquema de uma aplicação de um blog. No blog, os autores e os artigos seriam coleções e os comentários referentes aos artigos tornar-se-iam documentos inseridos nos documentos pertencentes à coleção de artigos (MURPHY; VOYER-PERRAULT, 2011). Desta maneira, temos que utilizar a referencia para selecionar as informações do autor de um determinado artigo. Para isso, devemos utilizar os comandos abaixo para a referência manual.

```
> artigo = db.artigos.findOne();
    {
        "_id"    : ObjectId("4b866f08234ae01d21d89604"),
        "autor"  : "João",
        "titulo"  : "MongoDB Article"
    }

> usuario = db.usuarios.findOne( { _id : artigo.autor } );

    {
        "_id"          : "João",
        "email"       : "joao@gmail.com",
        "endereço" : "Avenida Rio Branco",
        "telefone"   : "2520-2526"
    }
```

O primeiro comando seleciona um artigo da coleção artigos e atribui o resultado à variável artigo. Então esta variável contém os campos _id, autor e titulo. O segundo comando seleciona na coleção usuarios um usuário que possua no campo _id um valor igual ao campo autor da variável artigo. Com isso, são trazidas as informações do autor "João" que escreveu o artigo "MongoDB Article".

Além da forma manual, o MongoDB permite uma forma de referenciar outros documentos de uma maneira mais formal, o *DBRef*. *DBRefs* nada mais são que documentos contidos no próprio documento e que geralmente incluem o nome da coleção com um *ObjectID*. Desenvolvedores preferem utilizar o *DBRef* quando as coleções referenciadas diferem de um documento para outro, caso contrário a referência manual é mais eficiente (MERRIMAN; CHODROW, 2011). Cabe lembrar que para cada referência é necessária uma consulta na coleção referenciada para o retorno do objeto referenciado. Isso pode ter um alto custo para o desempenho caso a coleção referenciada seja muito grande. Para a criação desse tipo de referência é necessário a passagem de dois argumentos, da seguinte forma:

{ $ref : <nome da coleção>, $id : <valor do _id do documento referenciado> [, $db : <nome do database>] }

O valor do '_id' é um identificador único (que todos os documentos possuem). Este é um campo especial, que se não for inserido explicitamente, será criado pelo SGBD de forma a ser único no escopo da coleção.

2.4.2 Linguagem de Consulta

A linguagem de consulta utilizada pelo MongoDB é o BSON. BSON são documentos no formato binário do JSON. Esse documento BSON é gerado pelo MongoDB através dos documentos JSON enviados pelo cliente. Frequentemente as consultas que poderiam ser feitas com o SQL podem também ser expressas com o JSON (CHODROW; GILL, 2010).

JSON é um tipo de documento derivado do JavaScript para representação de estrutura de dados e objetos. Sendo assim é uma linguagem de marcação, como o XML. Apesar de não ser extensível como o XML, o JSON é considerado uma linguagem mais legível para humanos e mais eficiente para máquinas (ECMA INTERNATIONAL, 2006).

O documento JSON é construído sobre as seguintes estruturas:

- Coleção de pares chave/valor;
- Uma lista de valores ordenada.

Como exemplo de um documento JSON, temos:

{“cpf” : “129304229”, “nome”: “João”, “data_nascimento” : Date(‘1967,05,17’)}

Um exemplo de consulta em JSON, que pode ser enviado ao MongoDB, para que sejam selecionados os nomes e os cpf das pessoas é mostrado a seguir.

db.pessoas.find({}, {‘cpf’ : 1, ‘nome’: 1})

Nesse exemplo temos a utilização do comando *find,* que funciona de acordo com dois parâmetros. O primeiro representa o filtro que será utilizado na consulta. Como queremos retornar todos os dados da base, este parametro é passado um documento vazio mostrando que nenhum critério foi utilizado. Este primeiro parâmetro é análogo à cláusula *WHERE* do *SQL*. Já o segundo parâmetro representa quais os campos que devem ser retornados dos documentos selecionados, no caso o CPF e o nome das pessoas. Esse parâmetro é análogo ao *SELECT* do *SQL*.

Além do *find*, o MongoDB permite também a utilização de comandos de inserção, exclusão e atualização de dados. O comando de inserção é representado pelo *insert* que recebe como parâmetro o documento JSON que será inserido. Para inserir um documento novo, na coleção pessoas, contendo o nome e o cpf, utilizamos o comando a seguir.

db.pessoas.insert({“nome”: “João”, “cpf”: “1293040229”})

Para remover da coleção pessoas o documento cujo nome é João e o cpf é 1293040229, utilizamos o comando a seguir.

db.pessoas.remove({“nome”: “João”, “cpf” : “1293040229”})

Finalmente, o comando de atualizar é representado pelo *update* onde são passados quatro parâmetros. O primeiro representa o critério que estamos utilizando. Assim como no *find,* este parâmetro é análogo ao *WHERE* do *SQL*. O segundo parâmetro representa qual o objeto que será

atualizado seguido do valor. O terceiro parâmetro pode ser composto de *true* ou *false*, onde *true* representa que se o objeto não existir no documento o mesmo deverá ser criado. O quarto parâmetro também é composto de *true* ou *false*, onde *true* indica que todos os documentos que se encaixam nos critérios do primeiro parâmetro serão atualizados. Caso contrário, somente o primeiro documento encontrado no critério será atualizado. Desta maneira, para atualizarmos o cpf do primeiro documento encontrado cujo nome é igual a João, podemos utilizar o comando a seguir.

db.pessoas.update({"nome" : "João" } , { "cpf" : "1003040009" } , false , false)

São necessários modificadores de consultas para fazermos consultas por critérios. Esses modificadores podem ser operadores condicionais, ou mesmo operações como *in*, *exists*, dentre outras. Estes operadores são facilmente resolvidos pelo banco, pois são tratados como parte do documento BSON.

Um exemplo que utiliza estes modificadores pode ser visto na consulta a seguir. Nesta consulta buscamos filtrar todas as pessoas da base de dados que possuem o nome João ou Maria, através da utilização do modificador de consulta *in*. Além disso, a consulta também filtra as pessoas que possuem a chave CPF cadastradas no documento, através do modificador de consulta *exists*.

db.pessoas.find({'nome': {$in: ['João', 'Maria']}, 'cpf': {$exists : true}})

Entretanto, se uma consulta não conseguir ser representada utilizando estes modificadores, também é possível a utilização de consultas com as funções de *map/reduce* (DEAN; SANJAY, 2008). *Map/reduce* é uma ferramenta poderosa de processamento de dados e para agregações. Geralmente são mais complicadas de se escrever e normalmente sua utilização ocorre de forma análoga ao *group by* do *SQL*. No MongoDB, esta função funciona da seguinte forma: é selecionada uma coleção, processada segundo a função de *map/reduce* e então é retornada uma nova coleção com o resultado.

Apesar do map/reduce ser uma ferramenta poderosa, ela apresenta um desempenho ruim para resultados em tempo real. Este desempenho ruim é explicado pela limitação existente na avaliação do *map/reduce* feita pelo JavaScript. O motor do JavaScript permite apenas a execução de uma thread em paralelo, logo não aproveita os diversos núcleos que os processadores atuais possuem. Além da limitação de processamento, o *map/reduce* também não permite que outros processos sejam executados paralelamente a ele (HOROWITZ; STEARN, 2011).

2.4.3 Índices

Os índices no MongoDB funcionam de maneira muito semelhante aos índices nos bancos relacionais. Logo, a maioria dos conceitos e materiais sobre indexação para bancos relacionais também podem ser utilizados com o MongoDB (CHODROW; DIROLF, 2010).

O índice é uma estrutura de dados que coleta informações sobre o valor de um determinado campo do documento, implementada em Árvore B e utilizada pelo otimizador de consultas do MongoDB para rapidamente ordenar os documentos da coleção (MURPHY; MERRIMAN, 2011).

Quando uma única chave é utilizada na consulta, esta chave pode ser indexada para melhorar o desempenho da consulta. Entretanto, se o campo não está sendo chamado no método *find*, o índice não

irá melhorar a desempenho da consulta. Para criar um índice no MongoDB, deve-se utilizar o método *ensureIndex*, indicar quais campos serão indexados e indicar se o índice é ascendente (com o parâmetro 1) ou descendente (com o parâmetro -1). Para adicionar um índice no campo cpf do documento da Figura 14, deve ser utilizado o seguinte comando:

db.pessoas.ensureIndex({'cpf' : 1});

O MongoDB suporta chaves únicas, índices compostos, índices em qualquer campo do documento, inclusive nos documentos inseridos e índices geoespaciais. O MongoDB também permite índices esparsos, permitindo assim consultas utilizando o índice em coleções onde nem todos os documentos possuem o campo indexado, tendo apenas como limitação não possuir mais de um campo indexado.

Índices também podem ser utilizados para retornar o resultado de uma consulta, caso os campos consultados estejam indexados. Outra característica do MongoDB é que um índice único, por padrão, é criado no campo '_id'. Esse índice apenas não será criado se a coleção for do tipo *capped,* que é um tipo especial de coleção.

A desvantagem de criar índices é que este causa uma sobrecarga nas inserções, modificações e exclusões. Não é recomendada a indexação de todos os campos da coleção, pois além de processar a operação, nestes casos o SGBD precisa também alterar os índices.

O otimizador de consultas do MongoDB gera planos para todas as consultas que forem submetidas por um cliente, funcionando de uma forma diferente dos otimizadores tradicionais, que se baseiam no custo das consultas. Ao contrário destes modelos, o otimizador simplesmente executa diferentes planos em paralelo e aprende qual deles trabalha melhor, que é aquele que termina primeiro. Este método funciona bem para bancos não relacionais, pois não são realizadas junções, logo o espaço utilizado para os possíveis planos da consulta é muito menor.

Pode acontecer de um plano que funcionava bem, com o tempo apresentar um desempenho pior, com a consulta demorando mais a responder. O motivo dessa queda de desempenho ocorre devido a mudanças nos dados do banco ou mudanças nos valores dos parâmetros das consultas. Neste caso, o SGBD executará novamente diferentes planos em paralelo para verificar qual o mais adequado (MERRIMAN, 2010).

Outra característica do MongoDB é que caso seja necessário, o otimizador de consultas reordena os termos da consulta para obter as vantagens do índice. Por exemplo, dada a consulta {'x':'foo', 'y':'bar'} e o índice {"y":1, "x":1}, o MongoDB irá resolver por utilizar o índice (CHODROW; DIROLF, 2010).

2.5 REDIS

Redis é um banco de dados chave-valor. Ele também é referenciado como um servidor de estruturas de dados, uma vez que as chaves podem conter strings, listas, conjuntos e conjuntos ordenados. Esta é uma das características que fazem do Redis um banco diferente de muitos outros, também orientados a chave-valor, pois todo valor tem um tipo. Além disso, o Redis é um banco muito

rápido, pois ele carrega e armazena todos os dados em memória e as alterações são escritas em disco, em segundo plano.

2.5.1 Modelo de Dados

O Redis tem um modelo orientado a chave-valor. A essência deste modelo é a habilidade de armazenar dados, chamados de valor dentro de uma chave, como mostra a Figura 15. Este dado pode ser resgatado somente pela chave exata onde foi armazenado, ou seja, não existe uma busca por valor. O Redis funciona como um dicionário persistente. Todo valor armazenado possui um tipo (FINDLEY, 2010). Os tipos suportados são:

- Texto
- Lista
- Conjunto (*Set*)
- Conjunto Ordenado (*Sorted Set*)

O tipo de um valor determina quais operações (chamados de comandos) estão disponíveis para o valor em si (SANFILIPPO; NOORDHUIS, 2010). Por exemplo, para acrescentar elementos a uma lista armazenada na chave *x* é utilizado o comando *lpush* ou *rpush*. Na seção seguinte são mostrados exemplos de comandos que podem ser realizados para cada tipo de dados.

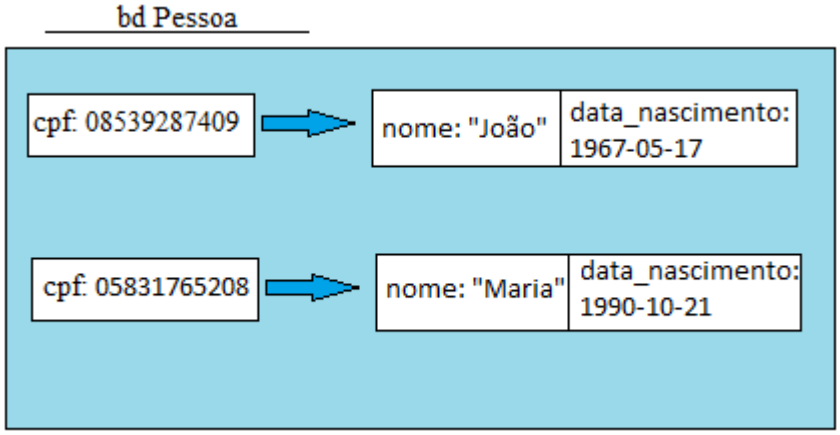

Figura 15: Modelo de dados do Redis.

2.5.2 Linguagem de Consulta

O Redis possui um *client* de consulta nativo, chamado redis-cli. Este é instalado junto com o SGBD. Como foi dito anteriormente, para cada tipo de valor existe uma operação específica que deve ser executada no banco. Abaixo são mostrados alguns exemplos mais utilizados (SANFILIPPO; NOORDHUIS, 2010).

- Comandos gerais:

- *Exists* <chave> – Testa se uma chave existe.
- *Del* <chave> – Apaga uma chave.
- *Type* <chave> – Retorna o tipo do valor armazenado na chave.
- *Key* <padrão> – Retorna todas as chaves que possuem o padrão passado.
- *Dbsize* – Retorna a quantidade de chaves presentes no banco.
- *Flushdb* – Remove todas as chaves do banco de dados corrente.

- Tipo Texto:
  - *Set* <chave> <valor> - Atribui um valor texto a uma chave.
  - *Get* <chave> - Retorna o campo texto da chave pesquisada.
  - *Getset* <chave> <valor> - Atribui um valor a uma chave e retorna o valor antigo da chave.
  - *Setnx* <chave> <valor> - Atribui um valor a uma chave se a chave não existir no banco.

- Tipo Lista:
  - *Rpush* <chave> <valor> - Adiciona um valor ao final da lista de uma chave.
  - *Lpush* <chave> <valor> - Adiciona um valor ao início da lista de uma chave.
  - *Llen* <chave> - Retorna o tamanho da lista de determinada chave.
  - *Lindex* <chave> <índice> - Retorna o elemento da lista na posição do índice citado.
  - *Lrange* <chave> <inicio> <fim> - Retorna um subconjunto de elementos de uma lista.
  - *Lpop* <chave> - Retorna e remove o primeiro elemento da lista.
  - *Rpop* <chave> - Retorna e remove o ultimo elemento da lista.

- Tipo Conjunto:
  - *Sadd* <chave> <membro> - Adiciona o membro ao valor do conjunto.
  - *Spop* <chave> <membro> - Remove e retorna o membro específico do valor do conjunto.
  - *Sunion* <chave1> <chave2>...<chave *n*> - Retorna a união entre os conjuntos armazenados nas chaves 1, chave2,..., chave *n*.

- *Smembers* <chave> - Retorna todos os membros do conjunto de uma determinada chave.

- Tipo Conjunto Ordenado:
  - *Zrem* <chave> <membro> - Remove o membro do conjunto
  - *Zadd* <chave> <posição> <membro> - Adiciona um membro ao conjunto. Caso já exista, somente a posição é alterada
  - *Zrange* <chave> <inicio> <fim> - Retorna um subconjunto dos elementos do conjunto ordenado.

- Ordenação:
  - *Sort* <chave> [*by* <padrão>][*limit* <quantidade>][*asc*|*desc*][*alpha*] – Ordena os elementos da lista, do conjunto ou do conjunto ordenado de determinada chave. Todas as cláusulas entre colchetes são opcionais.

- Comandos de persistência:
  - *Save* - Salva o banco em disco, de forma síncrona.

A seguir, estão representados alguns destes comandos, baseados no esquema montado na Figura 15. Os comandos *rpush* adicionam à chave cpf uma lista de valores: o nome e a data de nascimento de cada pessoa. Os comandos *lrange* selecionam os dois primeiros elementos da lista cuja chave é o cpf da pessoa.

>rpush 08539287409 Joao
>rpush 08539287409 1967-05-17
>lrange 08539287409 0 2
1) "Joao"
2) "1967-05-17"
>rpush 05831765208 Maria
>rpush 05831765208 1990-10-21
>lrange 05831765208 0 2
1) "Maria"
2) "1990-10-21"

2.5.3 Índices

O Redis, por padrão, não suporta a criação de índices. Caso o desenvolvedor necessite, ele pode modificar o código do Redis para incluir suporte à indexação. Esta implementação é facilitada pois o Redis é um projeto *open source*, ou seja, possui o código aberto para contribuições da comunidade (RUSSO, 2010).

Ao utilizar o Redis, deve-se lembrar que a organização dos dados deve ser feita de acordo com as consultas que serão realizadas. O Redis não implementa um otimizador de consultas. Ele provê primitivas muito rápidas. Entretanto, o desempenho da consulta é altamente dependente de como o usuário opta por organizar os dados (RUSSO, 2010).

2.6 CONSIDERAÇÕES FINAIS

Neste capítulo foram vistas as características de modelo dos dados relacionais e não relacionais. Além disso, discutimos como as consultas podem ser escritas, como funciona a criação de índices e o funcionamento do otimizador de consultas em cada um dos SGBDs.

No capítulo 3 é mostrado um esquema comum, retirado de uma ferramenta de *benchmark* e algumas consultas para mostrar o poder de expressão de cada um dos bancos selecionados. Além disso, também é explicada e exemplificada a dificuldade em realizar estas consultas.

3 EXEMPLO DE APLICAÇÃO

Como já vem sendo apresentado neste trabalho, os bancos de dados não relacionais ganharam bastante visibilidade e estão sendo utilizados por muitas empresas. Entretanto, diversas dúvidas surgem no assunto. Quando devemos utilizar esta tecnologia ao invés do modelo relacional tradicional? Que tipos de consultas esses SGBDs executam com facilidade? Qualquer modelo pode ser representado nessas novas tecnologias?

Neste capítulo, é utilizado um modelo relacional de uma corretora de valores. Este modelo é gerado por uma ferramenta de benchmark chamada TPC-E. Na seção 3.1 este modelo é explicado, assim como as consultas que serão realizadas.

Com este modelo pronto, realizamos uma série de consultas no MySQL, utilizando SQL. Estas consultas estão representadas na seção 3.2. Em seguida, nas seções seguintes, realizamos a transformação deste modelo relacional em modelos não relacionais para cada SGBD estudado e realizamos as mesmas consultas em cada um deles. Com este cenário conseguimos fazer uma comparação entre esses SGBDs nos quesitos modelagem e consulta.

3.1 TPC-E

O TPC-E é uma aplicação de benchmark composto por um conjunto de operações transacionais com o objetivo de exercitar as funcionalidades de sistemas complexos. O TPC-E centraliza suas operações em um modelo de atividades de uma corretora de valores, onde é necessário gerenciar contas de clientes, executar as ordens comerciais dos clientes e ser responsável pelas interações dos clientes com os mercados financeiros.

3.1.1 Estrutura a ser utilizada

O TPC-E gera a estrutura de um sistema de uma corretora de valores. Como esta é uma ferramenta de benchmark, ele gera a estrutura das tabelas e a preenche com uma grande quantidade de dados. Entretanto, durante a execução deste programa, tivemos que interromper o processamento, pois ele estava criando tabelas grandes que estavam estourando o espaço em disco da máquina que estava sendo utilizada para a carga. Como o nosso objetivo não é medir o desempenho de cada um dos bancos, decidimos utilizar apenas uma parte do modelo. Escolhemos a parte central do modelo, ou seja, as informações a respeito dos clientes e das contas destes clientes cadastrados nesta corretora.

A tabela *account_permission* possui as informações sobre permissão de acesso a uma determinada conta. Mais de uma pessoa pode realizar operações sobre uma determinada conta. Esta tabela é composta pelos atributos AP_CA_ID, que representa o identificador da conta do cliente, AP_ACL, que representa a lista de permissões que a pessoa possui sobre a conta do cliente, AP_TAX_ID, que representa o identificador da taxa da pessoa que possui acesso à conta do cliente e os atributos AP_L_NAME e AP_F_NAME que representam o nome e sobrenome desta pessoa.

A tabela *address* possui as informações a respeito de todos os endereços cadastrados no sistema. Eles podem ser endereços de clientes, companhias ou *exchange*. Esta tabela é composta pelos atributos AD_ID, que representa o identificador da lista de endereços, AD_LINE1 e AD_LINE2, que representam a linha 1 e a linha 2 do endereço, AD_ZC_CODE, que representa o CEP do endereço e o AD_CTRY, que representa o país do endereço.

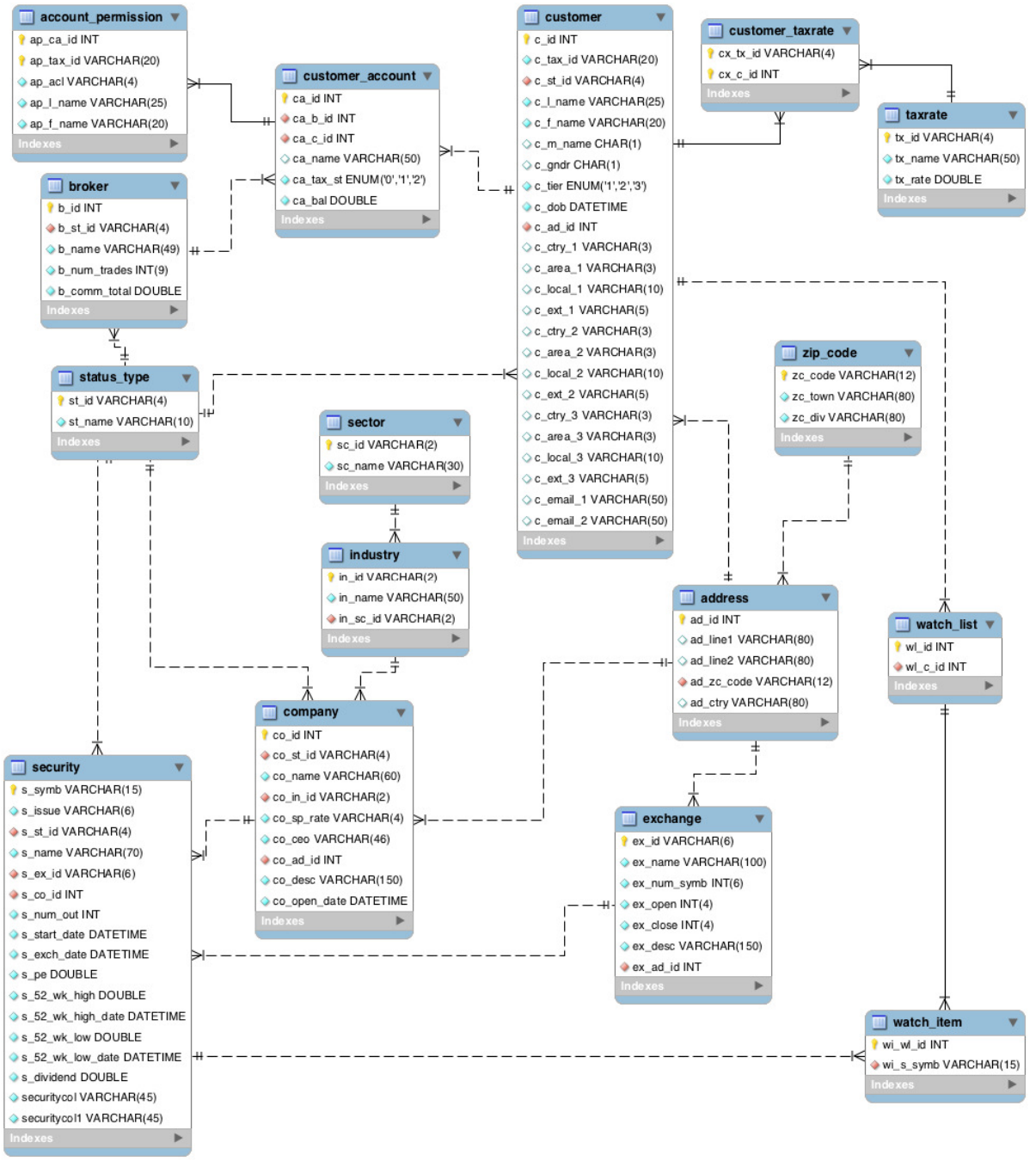

Figura 16: Modelo de Dados do TPC-E.

A tabela *broker* possui as informações dos agentes da corretora. Esta tabela é composta pelos atributos B_ID, que representa o identificador do *broker*, B_ST_ID, que representa o status do *broker* que pode ser ativo ou inativo, B_NAME, que representa o nome do broker, B_NUM_TRADES, que representa o número de transações que o *broker* realizou e B_COMM_TOTAL, que representa o valor total das comissões que o *broker* acumulou.

A tabela *company* possui informações sobre todas as empresas com títulos negociados publicamente. Esta tabela é composta pelos atributos CO_ID, que representa o identificador da

empresa, CO_ST_ID, que representa se a empresa é ativa ou não, CO_NAME, que representa o nome da empresa, CO_IN_ID, que representa o identificador da indústria à qual a empresa pertence, CO_SP_RATE, que representa a avaliação de crédito da empresa (feita pela Standard & Poor), CO_CEO, que representa o nome do chefe executivo da empresa, CO_AD_ID, que representa o identificador do endereço da empresa (referencia o identificador na tabela de endereços). Além disso, esta tabela também possui o atributo CO_DESC, que representa a descrição da empresa e CO_OPEN_DATE, que representa a data de fundação da empresa.

A tabela *customer* possui as informações de todos os clientes da corretora de valores. Esta tabela é composta pelos atributos C_ID, que representa o identificador do cliente, C_TAX_ID, que representa o identificador da taxa do cliente. Trata-se de um código alfanumérico usado externamente para comunicação com o cliente. O C_ST_ID é o identificador do status do cliente, C_F_NAME, C_M_NAME e C_L_NAME que representam o nome, nome do meio e sobrenome do cliente. Além disso, o cliente possui outras informações registradas, como o atributo C_GNDR, que representa o sexo da pessoa (M ou F), o atributo C_TIER, que indica o nível da conta do cliente. As taxas de contas nível 1 são as mais altas. Contas de nível 2 possuem taxas intermediárias, e contas de nível 3 pagam as taxas mais baratas.. O atributo C_DOB, que indica a data de nascimento do cliente, C_AD_ID, que é o identificador do endereço do cliente e referencia um registro da tabela *address*. O cliente pode cadastrar até 3 telefones distintos. Com isso, a tabela possui os atributos C_CTRY_1, C_CTRY_2 e C_CTRY_3 que indicam o código do país dos telefones, os atributos C_AREA_1, C_AREA_2 e C_AREA_3 que indicam o código de área dos telefones, os atributos C_LOCAL_1, C_LOCAL_2 e C_LOCAL_3 que indicam os telefones do cliente e os atributos C_EXT_1, C_EXT_2 e C_EXT_3 que representam o ramal dos números telefônicos. Finalmente, o cliente pode cadastrar 2 contatos eletrônicos que são representados pelos atributos C_EMAIL_1 e C_EMAIL_2 da tabela *customer*.

A tabela *customer_account* contém informações das contas de cada cliente. Esta tabela é composta pelos atributos CA_ID, que representa o identificador da conta do cliente, CA_B_ID, que representa o identificador do *broker* que gerencia esta conta, CA_C_ID, que representa o identificador do cliente que é o dono desta conta, CA_NAME, que representa o nome da conta do cliente, CA_TAX_ST, que representa se a conta pode ser taxada ou não e CA_BAL, que representa o saldo da conta. Os identificadores do cliente e do *broker* referenciam respectivamente as tabelas *customer* e *broker* que informam os detalhes de cada um.

A tabela *customer_taxrate* contém a relação entre os clientes e as taxas que eles têm que pagar. Esta tabela é composta pelos atributos CX_TX_ID e CX_C_ID, que representam respectivamente o identificador da taxa e o identificador do cliente que tem que pagar esta taxa. O identificador da taxa referencia a tabela *taxrate* e o identificador do cliente referencia a tabela *customer*.

A tabela *exchange* contém informações sobre corretoras. Ela possui como atributo o EX_ID, que representa o identificador da corretora, o EX_NAME, que representa o nome da corretora, EX_NUM_SYMB, que representa o número de títulos negociados nesta corretora, EX_OPEN, que representa o horário de abertura das transações diárias, EX_CLOSE, que representa o horário de términio das transações diárias, EX_DESC, que representa a descrição das transações e EX_AD_ID, que representa o identificador do endereço de correspondência.

A tabela *industry* contém informações sobre as indústrias. Elas são utilizadas para categorizar o tipo de indústria de uma determinada empresa. Por exemplo: banco, companhia aérea. Esta tabela é

composta pelo atributo IN_ID, que representa o identificador da indústria, IN_NAME, que representa o nome da indústria e IN_SC_ID, que representa o identificador do setor da indústria. O mesmo referencia a tabela *sector* que possui os detalhes de cada setor.

A tabela *sector* possui informações sobre os setores do mercado, por exemplo, energia, transporte, saúde. Esta tabela é composta pelos atributos SC_ID e SC_NAME, que representam o identificador da tabela de setores e o nome de cada setor, respectivamente.

A tabela *security* possui informações sobre cada título negociado em qualquer uma das bolsas de valores. Esta tabela é composta pelos atributos S_SYMB, que representa o identificador do título negociado, S_ISSUE, que representa o tipo de título, S_ST_ID, que representa o status do título (ativo ou inativo), S_NAME, que representa o nome do título, S_EX_ID, que representa em corretora o título é negociado, S_CO_ID, que representa o identificador da empresa que emitiu este título, S_NUM_OUT, que representa o número de ações deste título que estão em circulação. Além disso, esta tabela também possui os atributos atributos S_START_DATE, S_EXCH_DATE, S_52WK_HIGH_DATE e S_52WK_LOW_DATE, que indicam a data que o título começou a ser negociado e a data que o mesmo começou a ser negociado nesta corretora, a data da maior alta nas últimas 52 semanas, e a data da maior baixa das últimas 52 semanas, respectivamente. A tabela também é composta pelos atributos que indicam a cotação atual, o maior preço atingido nas últimas 52 semanas, e o menor preço atingido nas últimas 52 semanas, representados pelos atributos S_PE, S_52WK_HIGH e S_52WK_LOW, respectivamente. Finalmente, os atributos S_DIVIDEND e S_YIELD representam o dividendo anual e o percentual dos dividendos em relação ao preço do título.

A tabela *status_type* contém todos os tipos de status que são referenciados por diversas tabelas. Por exemplo: completo, pendente, cancelado. Esta tabela é composta pelos atributos ST_ID e ST_NAME que representam o identificador do status e o nome do mesmo.

A tabela *taxrate* possui as informações sobre as taxas cobradas. Esta tabela é composta pelos atributos TX_ID, TX_NAME e TX_RATE que representam respectivamente o identificador, nome e valor de cada taxa.

A tabela *watch_item* possui informações de quais títulos pertencem a cada lista de títulos criada. Essas listas são criadas, acompanhadas e analisadas pelos clientes. Esta tabela é composta pelos atributos WI_WL_ID e WI_S_SYMB. O primeiro é o identificador da *watch_list* e o segundo é o símbolo do título que está sendo acompanhado.

A tabela *watch_list* possui a relação de qual cliente criou determinada lista de títulos. Esta tabela é composta pelos atributos WL_ID e WL_C_ID que representam o identificador da lista de títulos e o identificador do cliente, respectivamente. O identificador de cliente referencia a tabela *customer* para indicar os detalhes de cada cliente.

A tabela *zip_code* possui as informações de CEP, cidade e país que serão referenciados pelos endereços. Esta tabela é composta pelos atributos ZC_CODE, ZC_TOWN e ZC_DIV que representam respectivamente o código postal, a cidade e o estado ou província dos endereços.

3.1.2 Consultas a serem realizadas

Tendo em vista a base que construímos, iremos realizar onze consultas em cada um dos SGBDs. Segue a explicação cada uma das consultas realizadas.

1. Selecionar o nome e sobrenome de todos os clientes (*customers)* cadastrados
2. Selecionar todas as informações dos títulos (*securities)* da *watch list* cuja chave primária é wl123.
3. Selecionar todos os *brokers* que estão com status cancelado.
4. Selecionar o setor em que atua a empresa (*company)* chamada Onemf.
5. Selecionar todas as empresas de New York.
6. Selecionar todos os clientes que possuem apenas uma conta associada.
7. Selecionar todas as empresas que foram abertas depois do ano de 2010.
8. Selecionar todas as corretoras (*exchange)* dos USA.
9. Selecionar todas as contas e permissões do cliente cujo identificador é c000.
10. Selecionar a soma dos valores das taxas que cada cliente tem que pagar.
11. Selecionar o nome de todos os *brokers* que gerenciam mais de uma conta de cliente (*customer account).*

Nas próximas seções são apresentadas a implementação deste modelo de dados em todos os SGBDs estudados.

3.2 MYSQL

3.2.1 Estrutura

A estrutura sugerida pelo TPC para a modelagem da corretora é um modelo relacional. Desta maneira, não fizemos alterações no modelo explicado na seção 3.1.1. Com isso, a estrutura aplicada ao MySQL é exatamente a mesma estrutura representada na Figura 16.

3.2.2 Consultas

Selecionar o nome e sobrenome de todos os cliente (*customers)* cadastrados. A primeira consulta realizada no MySQL é a mais simples. São selecionados apenas dois campos c_l_name e c_f_name da tabela *customer*, como mostra a Figura 17.

```
select c_l_name, c_f_name
from customer;
```

Figura 17: Selecionar o nome e sobrenome de todos os customers cadastrados, no MySQL.

Selecionar todas as informações das *securities* da *watch list* cuja chave primária é wl123. Já na segunda consulta, foi adicionado um complicador que é a utilização de junções para unir informações

de duas tabelas distintas. Neste caso, estão sendo selecionadas todas as informações das *securities* de uma determinada *watch_list*, no caso da Figura 18 é a *watch_list* wl123.

```
select s.*
from security s, watch_item wi
where s.s_symb = wi.wi_s_symb
and wi.wi_wl_id = wl123;
```

Figura 18: Selecionar, no MySQL, todas as informações das securities de uma determinada watch list.

Selecionar todos os *brokers* que estão com status cancelado. Na Figura 19 está representada a consulta realizada no MySQL para selecionar a todos os *brokers* que estão com o status cancelado. Para isso, selecionamos na tabela *broker* as tuplas que possuem o campo b_st_id preenchido com a palavra CNCL.

```
select *
from broker
where b_st_id = 'CNCL';
```

Figura 19: Selecionar todos os brokers que estão com status cancelado no MySQL.

Selecionar o setor em que atua a empresa (*company)* chamada Onemf. A próxima consulta a ser realizada é a consulta para selecionar o setor em que uma determinada empresa atua. Para isso, é necessário realizar operações de junções entre as tabelas *sector*, *industry* e *company*. Além disso, deve ser passado o nome da empresa que estamos interessados em selecionar, como pode ser verificado na Figura 20.

```
select s.sc_name
from sector s, industry in, company c
where s.sc_id = in.in_sc_id
and in.in_id = co.co_in_id
and co.co_name =  'Onemf';
```

Figura 20: Selecionar o setor de uma determinada company, no MySQL.

Selecionar todas as empresas em New York. Na Figura 21, é representada uma consulta que utiliza o operador semelhante às consultas anteriores. Selecionamos todas as empresas em New York. Para isso, realizamos junções nas tabelas *company*, *address* e *zip_code* e dizemos que a cidade desejada é New York.

```
select co.*
from company co, address ad, zip_code zc
where co.co_ad_id = ad.ad_id
and   ad.ad_zc_code  = zc.zc_code
and  zc_town = 'New York';
```

Figura 21: Selecionar todas as empresas em New York, no MySQL.

Selecionar todos os clientes que possuem apenas uma conta associada. Na próxima consulta representada na Figura 22, pode-se verificar que mais um operador foi adicionado à consulta: o operador de agregação. Neste, as tuplas são reunidas em grupos sobre os quais são aplicadas funções

agregadas. No nosso exemplo, foram reunidas as tuplas da coluna ca_c_id que representam o identificador de cada cliente e sobre ela foi calculada a quantidade de contas que o cliente possui.

Além disso, foi adicionado o operador de condição *having* que permite comparar o resultado obtido com esta cláusula, trazendo somente as tuplas que se adequam a esta condição, ou seja, só é selecionado o id do usuário que possui uma única conta associada.

```
select ca_c_id, count(*)
from customer_account
group by ca_c_id
having count(*) = 1;
```

Figura 22: Selecionar, no MySQL, todos os clientes que possuem apenas uma conta associada

Selecionar todas as empresas que foram abertas depois do ano de 2010. A consulta da Figura 23 é mais simples do que as anteriores. Nesta, selecionamos as informações de empresas que foram abertas a partir de 1º de Janeiro de 2010.

```
select *
from company
where co_open_date > '01/01/2010';
```

Figura 23: Selecionar, no MySQL, todas as companies que foram abertas depois do ano de 2010.

Selecionar todas as corretoras dos USA. Na Figura 24, selecionamos todas as corretoras que possuem como endereço o país USA. Para isso, realizamos junções entre as tabelas *exchange* e *address* e selecionamos somente as tuplas que possuem o USA como país.

```
select ex.*
from exchange ex, address ad
where ex.ex_ad_id = ad.ad_id
and  ad.ad_ctry = 'USA';
```

Figura 24: Selecionar todas corretoras dos USA, no MySQL.

Selecionar todas as contas e permissões do cliente cujo identificador é c000. Na consulta da Figura 25, trazemos todas as permissões das contas de um determinado cliente. Nesta consulta utilizamos um mecanismo de aninhamento do select, ou seja, executamos um select dentro da consulta para trazer somente as contas que pertencem ao cliente c000. Por fora, selecionamos as permissões das contas que estão sendo trazidas pelo select aninhado. Essa consulta também poderia ter sido escrita utilizando o operador de junções como foi utilizado anteriormente.

```
select *
from account_permission
where ap_ca_id in (
            select  ca_id
            from customer_account
            where ca.ca_c_id = 'c000'
);
```

Figura 25: Selecionar, no MySQL, todas as contas e permissões de um dado cliente.

Selecionar a soma dos valores das taxas que cada cliente tem que pagar. Na Figura 26 o operador de agregação foi utilizado mais uma vez. No nosso exemplo, foram reunidas as tuplas das colunas c_l_name e c_f_name que representam o nome e o sobrenome dos clientes e sobre elas foram calculadas as somas dos impostos que o cliente com este nome tem que pagar.

```
select c.c_l_name, c.c_f_name,  sum(tx.tx_rate) as valor_taxas
from customer c, customer_taxrate ctx, taxrate tx
where c.c_id = ctx.cx_c_id
and ctx.cx_tx_id = tx.tx_id
group by c.c_l_name, c.c_f_name;
```

Figura 26: Selecionar a quantidade de taxas cada cliente possui, no MySQL.

Selecionar o nome de todos os brokers que gerenciam mais de uma conta de cliente. Por fim, a última consulta também utiliza o operador de condição *having*. Como pode ser visto na Figura 27, a consulta traz somente os *brokers* que possuem mais de uma conta de cliente associada.

```
select ca.ca_b_id, b.b_name, count(*) as qntd_cust_account
from broker b, customer_account ca, customer c
where b.b_id = ca.ca_b_id
and ca.ca_c_id = c.c_id
group by ca.ca_b_id, b.b_name
having count(*) > 1;
```

Figura 27: Selecionar, no MySQL, o nome de todos os brokers que gerenciam mais de uma customer account.

3.3 SEDNA

3.3.1 Estrutura

No Sedna o modelo é representado através do XML Schema (W3C XML Schema). XML Schema permite uma definição de esquemas flexível e que permite a validação dos documentos XML.

Ao invés de criar apenas um documento com todas as informações, foram criados cinco tipos de documentos para representar o modelo. Esta abordagem foi feita de forma que os documentos mais importantes ficassem separados, assim como foi implementado nos outros SGBDs não relacionais. A seguir está representado o modelo dos clientes (*Customer*). Este modelo possui as informações do cliente, como nome, gênero, endereço, informções de suas taxas, contas e permissões.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:complexType name="tCustomer">
  <xs:sequence>
   <xs:element name="c_id" type="xs:integer"/>
   <xs:element name="c_l_name" type="xs:string"/>
   <xs:element name="c_f_name" type="xs:string"/>
   <xs:element name="c_m_name" type="xs:string" minOccurs="0" maxOccurs="1"/>
   <xs:element name="c_gndr" type="xs:string" minOccurs="0" maxOccurs="1"/>
```

```
    <xs:element name="c_tier" type="xs:integer"/>
    <xs:element name="c_dob" type="xs:date"/>
    <xs:element name="status_type" type="xs:string"/>
    <xs:element name="address" type="tAddress"/>
    <xs:element name="taxrate" type="tTaxRate" minOccurs="1" maxOccurs=" unbounded "/>
    <xs:element name="customer_accounts" type="tCustomerAccounts" minOccurs="0"
maxOccurs="unbounded"/>
    <xs:element name="watch_list" type="tWatchList" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="tAddress">
  <xs:sequence>
    <xs:element name="ad_line" type="xs:string" minOccurs="0" maxOccurs="1"/>
    <xs:element name="ad_line2" type="xs:string" minOccurs="0" maxOccurs="1"/>
    <xs:element name="ad_ctry" type="xs:string" minOccurs="0" maxOccurs="1"/>
    <xs:element name="zipcode" type="tZipCode"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="tZipCode">
  <xs:sequence>
    <xs:element name="zc_code" type="xs:string"/>
    <xs:element name="zc_town" type="xs:string"/>
    <xs:element name="zc_div" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="tTaxRate">
  <xs:sequence>
    <xs:element name="tx_id" type="xs:integer"/>
    <xs:element name="tx_name" type="xs:string"/>
    <xs:element name="tx_rate" type="xs:decimal"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="tCustomerAccounts">
  <xs:sequence>
    <xs:element name="ca_id" type="xs:integer"/>
    <xs:element name="ca_name" type="xs:string"/>
    <xs:element name="ca_tax_st" type="xs:integer"/>
    <xs:element name="ca_bal" type="xs:decimal"/>
    <xs:element name="broker_id" type="xs:integer"/>
    <xs:element name="account_permission" type="xs:tAccountPermission"/>
  </xs:sequence>
 </xs:complexType>
```

```
  <xs:complexType name="tAccountPermission">
    <xs:sequence>
      <xs:element name="ap_acl" type="xs:string"/>
      <xs:element name="ap_l_name" type="xs:string"/>
      <xs:element name="ap_f_name" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="tWatchList">
    <xs:sequence>
      <xs:element name="wl_id" type="xs:string"/>
      <xs:element name="id_security" type="xs:string" minOccurs="1" maxOccurs="unbounded"/>/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="tCustomers">
    <xs:sequence>
      <xs:element name="customer" type="tCustomer" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="customers" type="tCustomers"/>
</xs:schema>
```

Na Figura 28 é detalhado o modelo de dados do *Broker*. Este contém informações do broker como nome, número de transações, quantidade total das comissões e seu status.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="tBroker">
    <xs:sequence>
      <xs:element name="id" type="xs:integer"/>
      <xs:element name="b_name" type="xs:string"/>
      <xs:element name="b_num_trades" type="xs:integer"/>
      <xs:element name="b_comm_total" type="xs:decimal"/>
      <xs:element name="status_type" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="tBrokers">
    <xs:sequence>
      <xs:element name="broker" type="tBroker" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="brokers" type="tBrokers"/>
</xs:schema>
```

Figura 28: XML Schema do documento broker

Na Figura 29 é representado o modelo de *Company*, onde estão cadastradas algumas informações como nome, descrição, data de abertura, endereço, crédito da empresa entre outras.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:complexType name="tCompany">
  <xs:sequence>
   <xs:element name="id" type="xs:string"/>
   <xs:element name="co_name" type="xs:string"/>
   <xs:element name="co_sp_rate" type="xs:string"/>
   <xs:element name="co_desc" type="xs:string"/>
   <xs:element name="co_open_date" type="xs:date"/>
   <xs:element name="status_type" type="xs:string"/>
   <xs:element name="industries" type="tIndustries"/>
   <xs:element name="address" type="tAddress"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="tIndustries">
  <xs:sequence>
   <xs:element name="in_name" type="xs:string"/>
   <xs:element name="sector" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="tAddress">
  <xs:sequence>
   <xs:element name="ad_line" type="xs:string" minOccurs="0" maxOccurs="1"/>
   <xs:element name="ad_line2" type="xs:string" minOccurs="0" maxOccurs="1"/>
   <xs:element name="ad_ctry" type="xs:string" minOccurs="0" maxOccurs="1"/>
   <xs:element name="zipcode" type="tZipCode"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="tZipCode">
  <xs:sequence>
   <xs:element name="zc_code" type="xs:string"/>
   <xs:element name="zc_town" type="xs:string"/>
   <xs:element name="zc_div" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="tCompanies">
  <xs:sequence>
   <xs:element name="company" type="tCompany" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
 <xs:element name="companies" type="tCompanies"/>
</xs:schema>
```

Figura 29: XML Schema do documento company

Na Figura 30 é representado o modelo de dados de *Exchange*. Neste modelo estão representados os elementos que indicam o nome, a data de abertura e fechamento, endereço e o número de títulos negociados na corretora. O endereço é definido por um tipo composto pelos elementos logradouro, estado e o CEP. Este também é definido como um tipo que contém atributos

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:complexType name="tExchange">
  <xs:sequence>
   <xs:element name="id" type="xs:string"/>
   <xs:element name="ex_name" type="xs:string"/>
   <xs:element name="ex_num_symb" type="xs:integer"/>
   <xs:element name="ex_open" type="xs:string"/>
   <xs:element name="ex_close" type="xs:string"/>
   <xs:element name="address" type="tAddress"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="tAddress">
  <xs:sequence>
   <xs:element name="ad_line" type="xs:string" minOccurs="0" maxOccurs="1"/>
   <xs:element name="ad_line2" type="xs:string" minOccurs="0" maxOccurs="1"/>
   <xs:element name="ad_ctry" type="xs:string" minOccurs="0" maxOccurs="1"/>
   <xs:element name="zipcode" type="tZipCode"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="tZipCode">
  <xs:sequence>
   <xs:element name="zc_code" type="xs:string"/>
   <xs:element name="zc_town" type="xs:string"/>
   <xs:element name="zc_div" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="tExchanges">
  <xs:sequence>
   <xs:element name="exchange" type="tExchange" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>

 <xs:element name="exchanges" type="tExchanges"/>

</xs:schema>
```

Figura 30: XML Schema do documento exchange

Na Figura 31 é representado o modelo de *Security*. Neste modelo é possível verificar algumas informações de cada título negociado, como nome, empressa associada, status, a data da negociação, quantidade, o valor de alta e de baixa de cada título entre outras características.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:complexType name="tSecurity">
  <xs:sequence>
   <xs:element name="id" type="xs:string"/>
   <xs:element name="s_issue" type="xs:string"/>
   <xs:element name="s_name" type="xs:string"/>
   <xs:element name="s_ex_id" type="xs:string"/>
   <xs:element name="s_co_id" type="xs:string"/>
   <xs:element name="s_num_out" type="xs:integer"/>
   <xs:element name="s_start_date" type="xs:date"/>
   <xs:element name="s_exch_date" type="xs:date"/>
   <xs:element name="s_pe" type="xs:integer"/>
   <xs:element name="s_52wk_high" type="xs:integer"/>
   <xs:element name="s_52wk_high_date" type="xs:date"/>
   <xs:element name="s_52wk_low" type="xs:integer"/>
   <xs:element name="s_52wk_low_date" type="xs:date"/>
   <xs:element name="s_dividend" type="xs:decimal"/>
   <xs:element name="s_yield" type="xs:decimal"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="tSecurities">
  <xs:sequence>
   <xs:element name="security" type="tSecurity" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
 <xs:element name="securities" type="tSecurities"/>
</xs:schema>
```

Figura 31: XML Schema do documento security

3.3.2 Consultas

Selecionar o nome e sobrenome de todos os clientes cadastrados. A Figura 32 mostra a primeira consulta no Sedna, expressa em XQuery, onde todos os nomes dos clientes são retornados. Para isso é necessário um iterador. Esse iterador percorre todos os elementos do documento de maneira seqüencial. Desta maneira, este verifica se cada um dos elementos satisfazem às condições de nossa consulta.

```
<customers>
{
for $x in doc("customer.xml")/customers/customer
return
 <customer>
 {$x/c_l_name}
 {$x/c_f_name}
 </customer>
}
</customers>
```

Figura 32: Consulta do nome dos customers no Sedna.

Selecionar todas as informações das securities da watch list cujo identificador é wl123. Na segunda consulta (Figura 33) para retornarmos os *securities* de uma determinada *watch_list* precisamos utilizar junções através do s_id encontrado no *watch_list* dentro da coleção do *customer*.

```
<security_list>
{
    for $w in doc("customer.xml")//customer/watch_list[wl_id =wl123],
          $s in doc("security.xml")//security
         where $s/id = $w/id_security
         return $s
}
</security_list>
```

Figura 33: Consulta no Sedna com junção para retornar a lista de securities.

Selecionar todos os *brokers* que estão com status cancelado. Utilizamos a consulta da Figura 34 para retornar os *brokers* que estão com o seu status como cancelado.

```
for $br in doc("broker.xml")//broker
where $br/status_type='CNCL'
return $br
```

Figura 34: Consulta no Sedna para o retorno dos brokers filtrando pelo status.

Selecionar o setor em que atua a empresa (*company)* chamada Onemf. A quarta consulta, representada na Figura 35, retorna em qual setor (ou setores) certa empresa está. Pelo modelo que foi utilizado para banco de dados do Sedna, não são necessárias as junções que foram necessárias no modelo relacional.

```
for $x in doc('company.xml')//company
where $x/co_name = 'Onemf'
return $x/industries/sector
```

Figura 35: Consulta que retorna qual setor da indústria, certa companhia está no Sedna.

Selecionar todas as empresas em New York. Na quinta consulta realizada, procuramos as empresas que estão localizadas em New York. Na Figura 36 temos a XQuery realizada para busca dessas empresas.

```
for $x in doc('company.xml')//company
where $x/address/zipcode/zc_town = 'New York'
return $x
```

Figura 36: Consulta em XQuery que retorna as empresas localizadas em New York.

Selecionar todos os clientes que possuem apenas uma conta associada. Para a sexta consulta realizada, que busca os clientes que possuem apenas uma conta associada, utilizamos a função *count*, como mostrado na Figura 37.

```
for $x in doc('customer.xml')
where (count($x//customer_accounts) = 1)
return $x
```

Figura 37: Consulta que retorna os clientes com apenas uma conta associada.

Selecionar todas as empresas que foram abertas depois do ano de 2010. Utilizamos a consulta da Figura 38 para retornar as companhias que foram abertas após o inicio do ano de 2010, para isso é necessário um filtro XPATH.

```
for $x in doc('company.xml')//company[co_open_date > '2010-1-1']
return $x
```

Figura 38: Consulta em XQuery que retorna as companhias abertas após 01/01/2010.

Selecionar todas as corretoras dos USA. A consulta da Figura 39 é utilizada para retornar as corretoras que estão localizadas nos Estados Unidos, como o endereço de cada transação está no próprio documento, permite uma consulta sem junções.

```
for $x in doc('exchange.xml')//exchange
where $x/address/ad_ctry = 'USA'
return $x
```

Figura 39: Consulta em XQuery que retorna as exchanges que estão localizadas nos EUA.

Selecionar todas as contas e permissões do cliente cujo identificador é c000. A consulta da Figura 40 é utilizada para retornar os detalhes das contas de um determinado cliente.

```
for $x in doc('customer.xml')//customer[id ='c000']
return $x/customer_account
```

Figura 40: Consulta que retorna os detalhes das contas de um determinado cliente.

Selecionar a soma dos valores das taxas que cada cliente tem que pagar. A consulta da Figura 41 é utilizada para retornar a soma das taxas dos clientes, não sendo necessário utilizar uma junção.

```
for $x in doc("customer.xml")//customer
return
<customers>
{$x/c_l_name}
{$x/c_f_name}
<qntd_taxas>{sum($x/taxrate/tx_rate) }</qntd_taxas>
</customers>
```

Figura 41: Consulta que retorna a soma das taxas de um customer.

Selecionar o nome de todos os brokers que gerenciam mais de uma conta de cliente. Na Figura 42 está representada a última consulta realizada no Sedna, para descobrir o nome dos *brokers* que gerenciam mais de uma conta, utilizando a função *count* e um operador condicional.

```
for $x in doc("brokers.xml")//broker/id
let $y := doc("customers.xml")//customer[customer_accounts/broker_id=$x]
where $y > 1
return $y/../broker_name
```

Figura 42: Consulta retornando os nomes de brokers com mais de uma conta associada.

3.4 CASSANDRA

3.4.1 Estrutura

No Cassandra, o modelo é representado através de cinco famílias de colunas, são elas: *Customer*, *Broker*, *Customer Account*, *Exchange*, *Security*, *Taxrate*, *Watch List* e *Company*. Nas imagens da Figura 43 até a Figura 50 são representados apenas o exemplo de uma linha de cada família.

Todas as famílias são formadas por um conjunto de chaves que referenciam um conjunto de colunas. As chaves são escolhidas pelo desenvolvedor. Podem ser um valor incremental, ou apenas um identificador, mas devem ser um valor único dentro da família pois representam o identificador de cada grupo de coluna. A chave, quando comparada ao modelo relacional, é análoga à chave primária das tabelas. Cada chave possui um grupo de colunas, que podem ou não ser iguais para todas as chaves. Cada coluna possui atribuída a ela, um valor texto ou inteiro.

No modelo relacional, o cliente é identificado pelo campo c_id que é a chave primária da tabela. Esse campo é composto pelos valores c000, c001 e assim por diante, de maneira incremental. No caso das nossas famílias de colunas, as chaves foram extraídas deste modelo relacional, ou seja, a chave c000 é um registro da chave primária da tabela *Customer* e o mesmo foi feito para todas as outras famílias. O Cassandra permite que a estrutura seja criada incialmente, mesmo que nenhum dado exista na base de dados. Dessa maneira, criamos a estrutura de cada família com todas as possíveis colunas e em seguida, inserimos os dados que foram extraídos do modelo relacional.

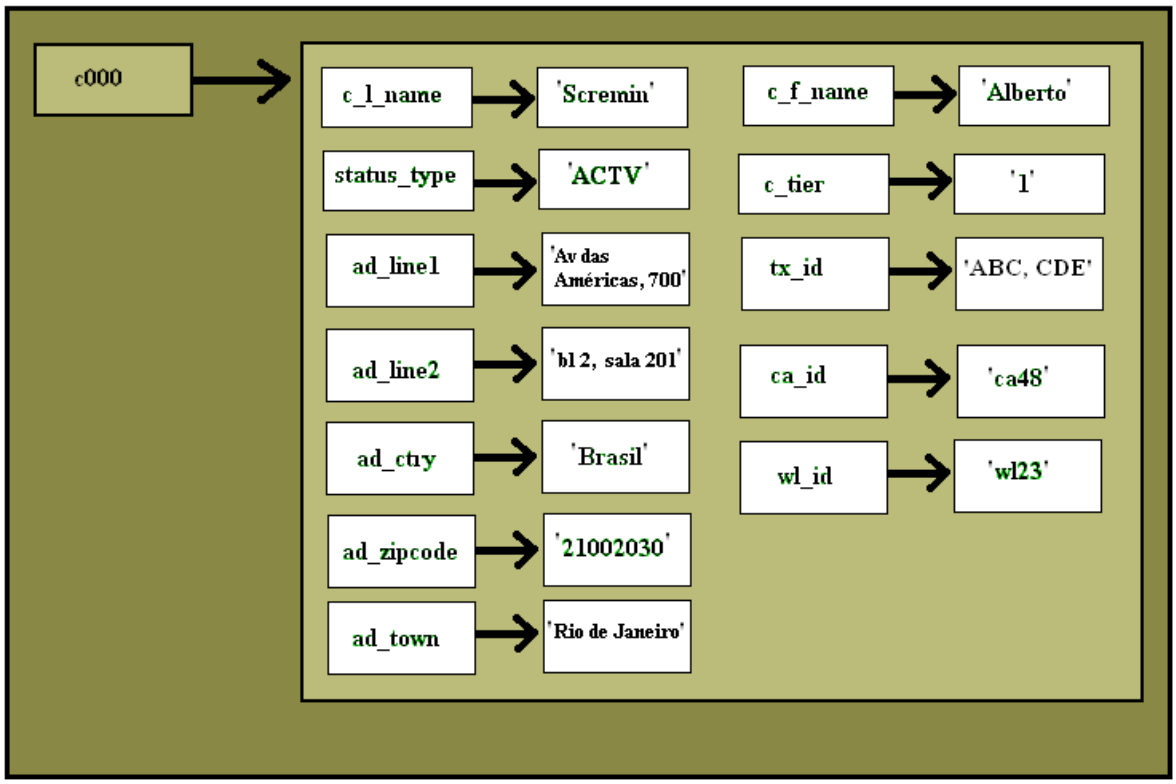

Figura 43: Família de Coluna Customer.

A Figura 44 representa a família *Broker*. Esta família é composta por um conjunto de linhas onde cada linha possui uma chave. Esta chave, assim como no modelo *customer*, é resultado da transformação da chave primária da tabela *broker* do modelo relacional. Além disso, cada chave possui um conjunto de colunas. As colunas deste modelo representam as características dos *brokers* e também existe uma coluna que representa a lista de *customer_account* que cada *broker* gerencia.

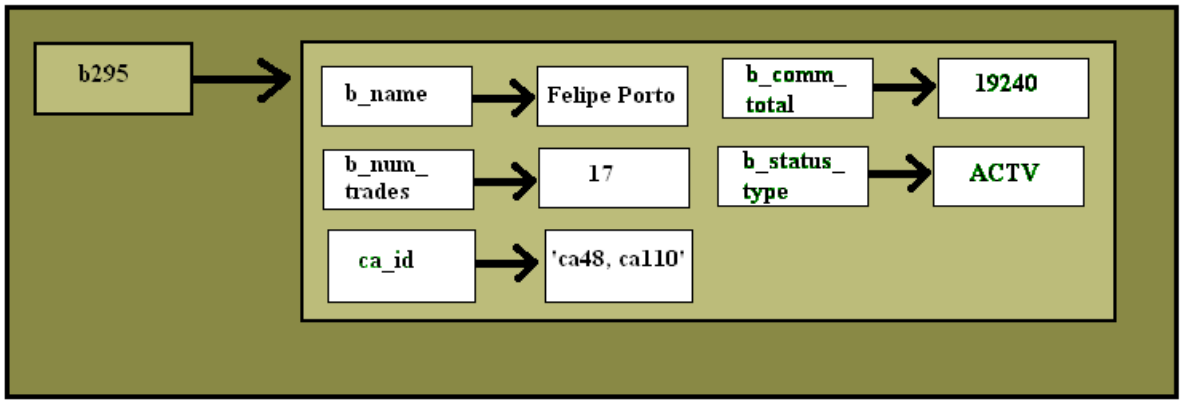

Figura 44: Família de Coluna Broker.

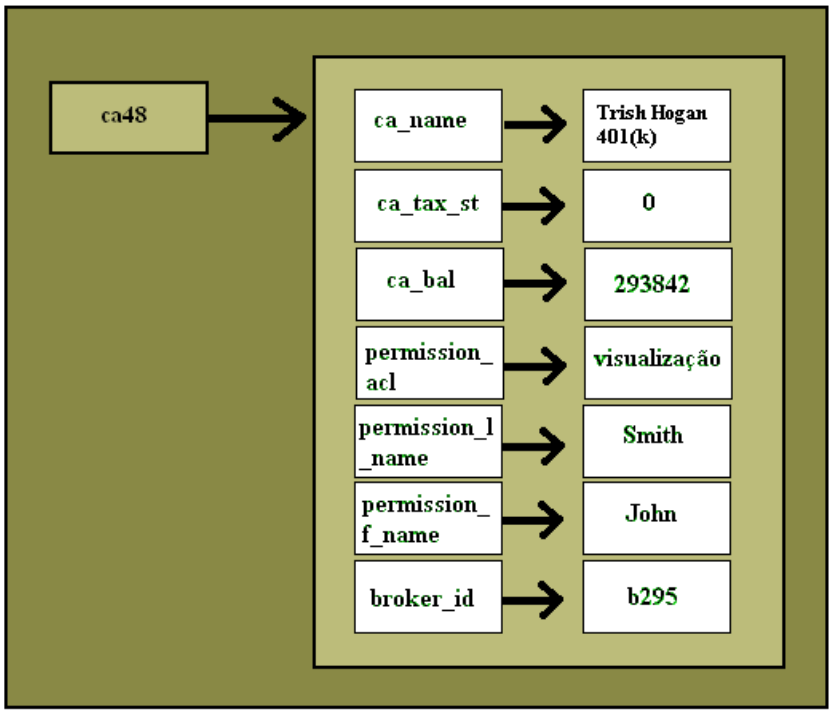

Figura 45: Família de Coluna Customer Account.

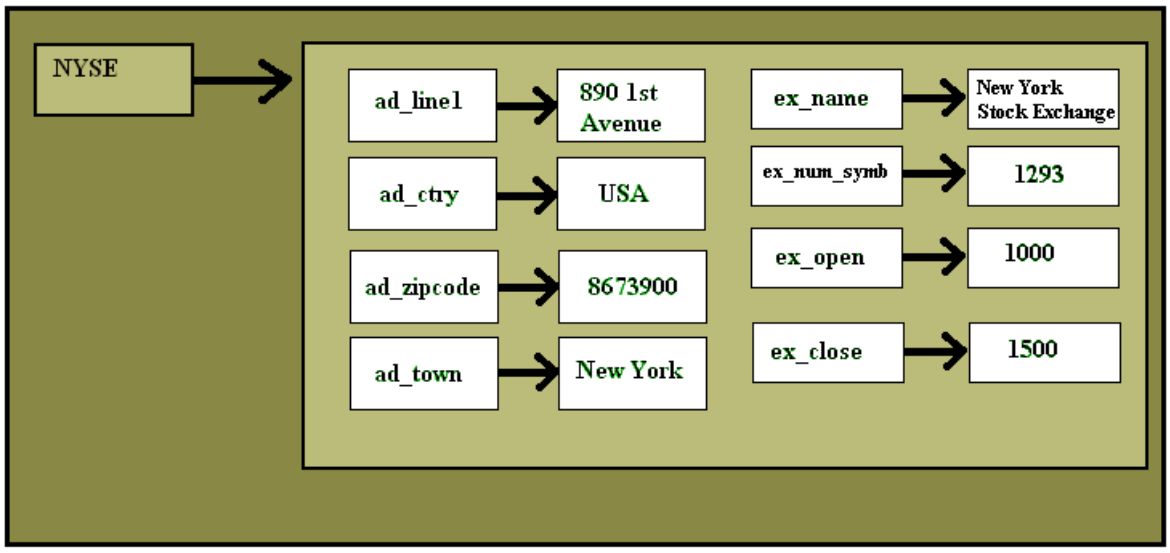

Figura 46: Família de Coluna Exchange

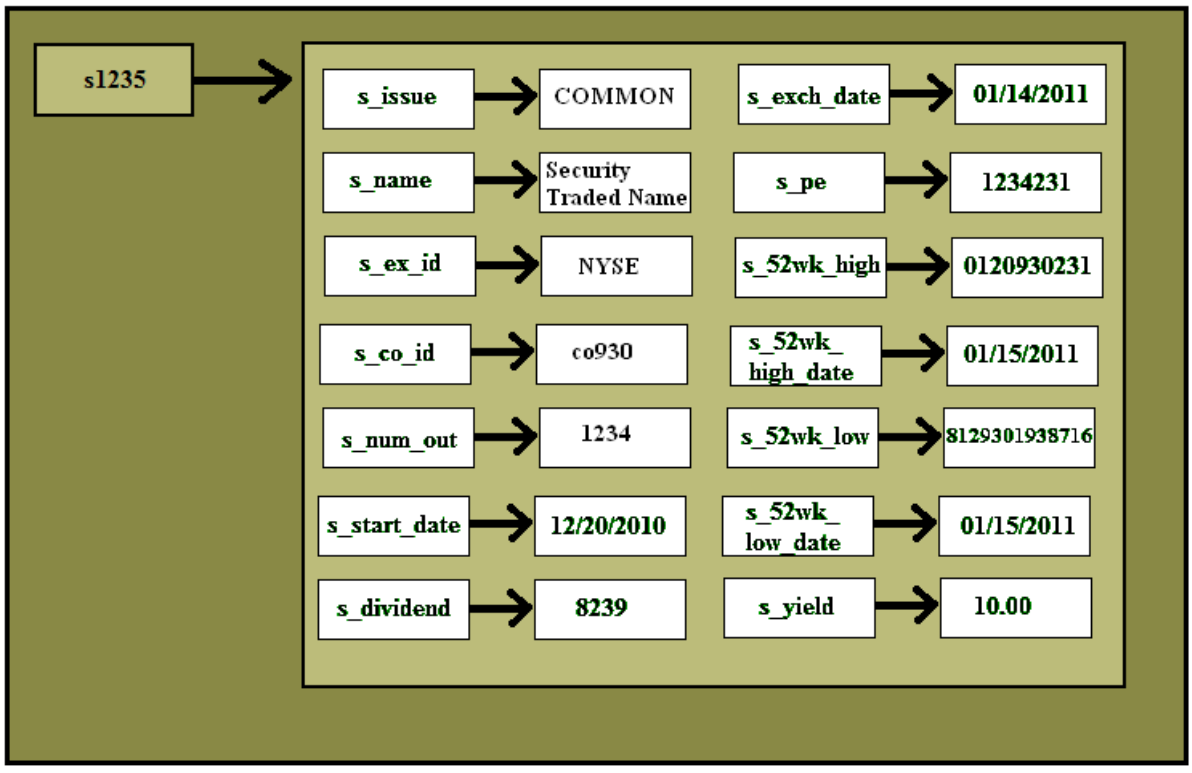

Figura 47: Família de Coluna Security

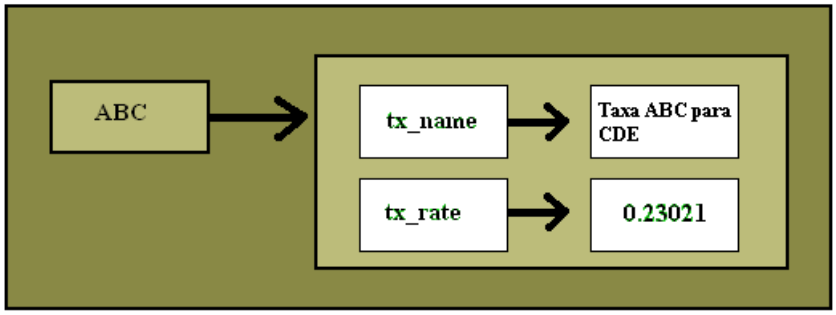

Figura 48: Família de Coluna Taxrate

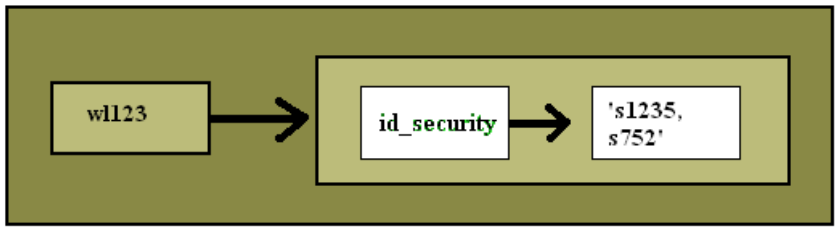

Figura 49: Família de Coluna Watch_list

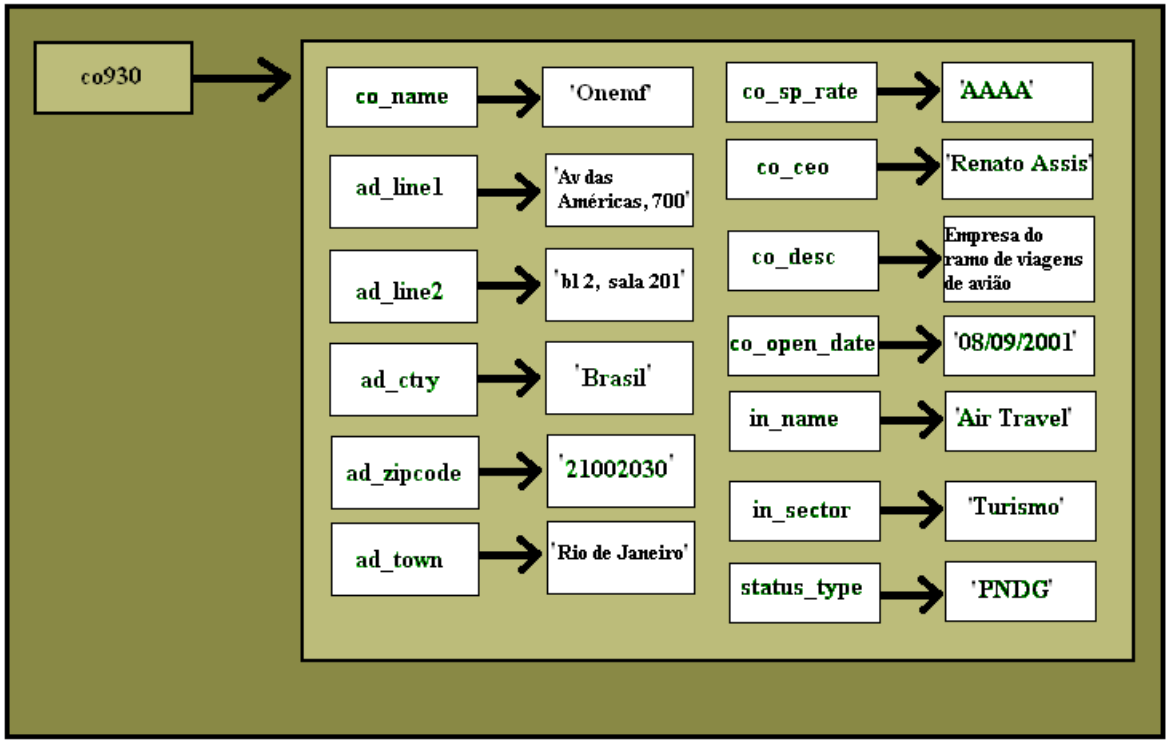

Figura 50: Família de Coluna Company.

3.4.2 Consultas

Selecionar o nome e sobrenome de todos os cliente cadastrados. A primeira consulta realizada no Cassandra está representada na Figura 51. O Cassandra não permite a criação de laços de repetição, então, para trazer o resultado esperado, é necessário realizar este laço na aplicação. Criamos o código em *python* que também está representado na Figura 51 para simular este laço de repetição.

Primeiramente, utilizamos o comando *list*, para listar todas as chaves da família *customer*. Esta listagem é utilizada no laço para ler uma chave de cliente distinta até que todas as chaves sejam lidas. Sendo assim, a cada iteração iremos executar os comandos de *get* mostrados. Este comando seleciona na família *customer*, inicialmente na chave c000, o nome e o sobrenome de cada cliente.

```
lista_customer_id = list customer;
for customer_id in lista_de_customer_id:
    get customer ['customer_id'] ['c_l_name'];
    get customer ['customer_id'] ['c_f_name'];
```

Figura 51: Consulta do nome dos customers no Cassandra.

Selecionar todas as informações das *securities* da *watch list* cujo identificador é wl123. Na segunda consulta, encontramos mais um obstáculo. O Cassandra não suporta a criação de listas no campo valor, ele só permite o armazenamento de campos textos e inteiros. Com isso, o que tratávamos como uma lista teve que ser inserido em um campo texto.

Sendo assim, na família *watch_list* temos para cada identificador de *watch list* uma string contendo o identificador de cada *security*, separados por vírgula. O primeiro *get* da Figura 52 foi utilizado para trazer esta lista de *id* de *securities* da *watch list* wl123. Mais uma vez devemos tratar na aplicação esse laço de repetição. Este procedimento é demonstrado no código *python* da Figura 52. Já os comandos *get* em seguida foram realizados para cada identificador selecionado anteriormente. Eles trazem todas as informações da família *security*.

```
lista_security := get watch_list ['wl123'] ['id_security'];
for security_id in lista_security:
    get security ['security_id'];
```

Figura 52: Selecionar todas as informações das securities de uma determinada watch list no Cassandra.

Selecionar todos os brokers que estão com status cancelado. A próxima consulta realizada foi representada de uma maneira simples com apenas um comando no Cassandra. Como mostrado na Figura 53, executamos um *get* na família de coluna *broker*, onde a coluna *b_status_type* está preenchida como cancelado.

```
get broker where b_status_type = 'CNCL';
```

Figura 53: Selecionar todos os brokers que estão com status cancelado no Cassandra.

Selecionar o setor em que atua a empresa (*company)* chamada Onemf. A quarta consulta também pode ser representada de uma maneira simples. Como mostrado na Figura 54, utilizamos o comando *get* que seleciona todas as colunas da família *company*, inclusive a coluna que desejamos (in_sector). O comando *get* utilizado garante que somente a linha que possuir a coluna co_name igual à Onemf será retornada.

```
get company where co_name = 'Onemf';
```

Figura 54: Selecionar o setor de uma determinada company no Cassandra.

Selecionar todas as empresas em New York. A próxima consulta, representada na Figura 55, seleciona todas as empresas que possuem na coluna *ad_town*, o dado New York.

```
get company where ad_town = 'New York';
```

Figura 55: Selecionar, no Cassandra, todas as empresas em New York.

Selecionar todos os clientes que possuem apenas uma conta associada. A consulta representada na Figura 56, apesar de simples, também requer um tratamento da aplicação. Utilizamos o laço de repetição para ler as chaves de todos os clientes. Além disso, devemos realizar uma contagem da quantidade de contas que o cliente possui e exibir somente as informações dos clientes que possuem uma única conta.

```
lista_customer_id = list customer;
for customer_id in lista_customer_id:
    c_ca = get customer ['customer_id'] ['ca_id'];
    c_ca_qtd = len(c_ca)
    if c_ca_qtd = 1:
        get customer ['customer_id'];
```

Figura 56: Selecionar, no Cassandra, todos os clientes que possuem apenas uma conta associada

Selecionar todas as empresas que foram abertas depois do ano de 2010. Na próxima consulta, desejamos selecionar todas as empresas que foram abertas após o ano de 2010. Para isso, conforme mostrado na Figura 57, realizamos um *get* na família *company*, onde a coluna *co_open_date* é maior ou igual à data 01/01/2010.

```
get company where co_open_date >= '01/01/2010';
```

Figura 57: Selecionar todas as companies que foram abertas depois do ano de 2010 no Cassandra.

Selecionar todas as corretoras dos USA. A consulta da Figura 58 é realizada de maneira muito semelhante às anteriores. Na família *exchange* são selecionadas todas as linhas que possuem *ad_ctry* cadastrados como USA.

```
get exchange where ad_ctry = 'USA';
```

Figura 58: Selecionar, no Cassandra, todos exchange dos USA

Selecionar todas as contas e permissões do cliente cujo identificador é c000. Para a próxima consulta, também foi utilizado um código para a iteração. Dado um determinado *customer*, utilizamos o *get* para selecionar a lista de contas que este cliente possui. Em seguida, para cada conta de cliente, utilizamos outro *get* para trazer as informações desta conta. Como pode ser visto na Figura 59, representamos com um código em *python* as instruções que devem ser implementadas na aplicação e os *get* que são executados no Cassandra.

```
customer_account_id = get customer ['c000'] ['ca_id'];
for cca_id in customer_account_id:
    get customer_account ['cca_id'];
```

Figura 59: Selecionar, no Cassandra, todas as contas e permissões de um dado cliente

Selecionar a soma dos valores das taxas que cada cliente tem que pagar. Na Figura 60, representamos mais uma consulta. Nesta são utilizados dois laços de iteração, primeiramente utilizamos um para selecionar todos os *ids* dos clientes e utilizamos o comando *get* para trazer da família *customer* os identificadores das taxas de cada cliente. Em seguida, é utilizado outro laço para cada taxa de um determinado cliente e outro *get* para trazer da família *taxrate*, os valores das taxas. Finalmente, utilizamos um acumulador para realizar a soma desses valores e exibir o cliente e o valor total da taxa que o mesmo deve pagar.

```
lista_customer_id = list customer;
for customer_id in lista_customer_id:
        lista_tx_id = get customer [‘customer_id][‘tx_id’];
        for tx_id in lista_tx_id:
                tx_val = get taxrate [‘tx_id’] [‘tx_rate’];
                val_tx_tot = val_tx_tot + tx_val
        print customer_id, ‘possui’, val_tx_tot
```

Figura 60: Selecionar, no Cassandra, a quantidade de taxas cada cliente possui.

Selecionar o nome de todos os *brokers* que gerenciam mais de uma conta de cliente. Na última consulta representada na Figura 61, obtivemos os mesmos obstáculos já mencionados nas consultas anteriores. Mais uma vez foi necessária a criação do laço de repetição na aplicação, desta vez para selecionar todos os *brokers* cadastrados. Para cada *broker*, selecionamos as contas que eles gerenciam.

Finamente, para obter o resultado esperado, devemos testar na aplicação se o *broker* possui mais de uma conta de cliente associada. O requisito desta consulta é selecionar somente os *brokers* que possuem mais de uma conta de cliente associada. Desta maneira, somente os registros que obedecerem a esta condição devem ser mostrados.

```
lista_broker_id = list broker;
for broker_id in lista_broker_id :
        b_ca_id = get broker [‘broker_id] [‘ca_id’];
        qtd_b_ca = len(b_ca_id)
        if qtd_b_ca > 1 :
                get broker [‘broker_id’] [‘b_name’];
```

Figura 61: Selecionar, no Cassandra, o nome dos brokers que gerenciam mais de uma customer_account.

3.5 MONGODB

3.5.1 Estrutura

No MongoDB as tabelas foram representadas através de cinco coleções, que são as seguintes: *customer*, *broker*, *company*, *exchange* e *security*. Escolhemos essa estrutura por representar melhor o banco existente no modelo relacional, de forma a respeitar os princípios básicos oferecidos pelos documentos do MongoDB.

Desta forma, as tabelas que se mostraram mais importantes em nosso modelo tornaram-se coleções, enquanto as outras tabelas tornaram-se documentos inseridos nas coleções que se relacionavam diretamente. As referências utilizadas para os documentos que não puderam tornar-se

documentos inseridos foram manuais, deixando a cargo do desenvolvedor o tratamento no caso de junções entre documentos do banco. Esse tipo de referência foi utilizado para referenciar as coleções *exchange* e *company* pela coleção de *security*.

A coleção customer apresenta as informações do cliente e também contém informações como seu endereço, taxas e suas contas. Contém também informações relativas à segurança, tendo como referência um identificador da coleção *security*. Na Figura 62 é mostrado um exemplo desse tipo de documento.

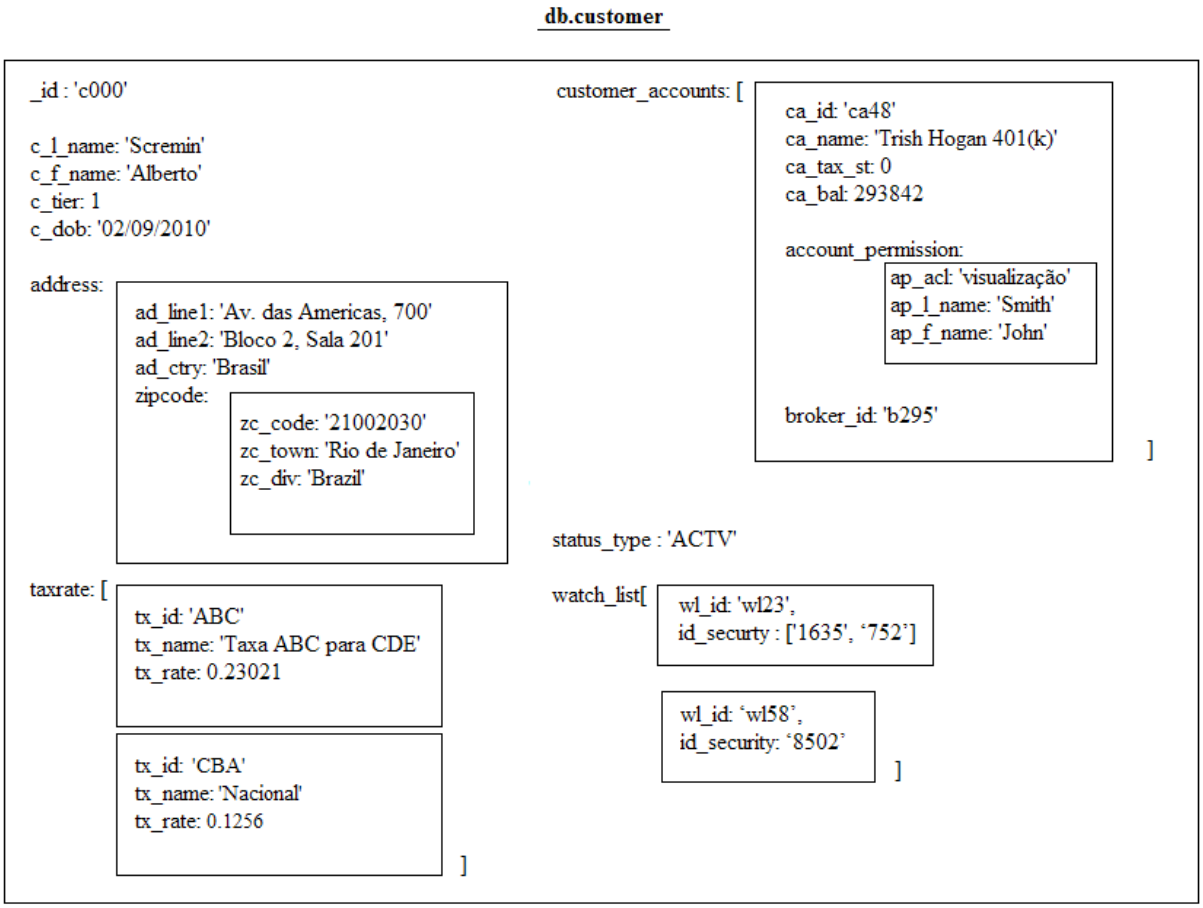

Figura 62: Exemplo de documento da coleção customer.

Outro elemento que consideramos como um documento no modelo foi o *broker*, representando as informações que o *broker* possui na Figura 63. O broker pode possuir como informações seu nome, o número de transações feitas, o valor total delas e o seu status.

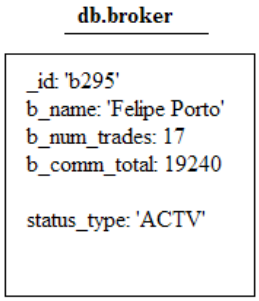

Figura 63: Exemplo de documento da coleção broker.

A coleção da companhia representada na Figura 64 engloba documentos sobre suas características, endereço, além do tipo de indústria e setor.

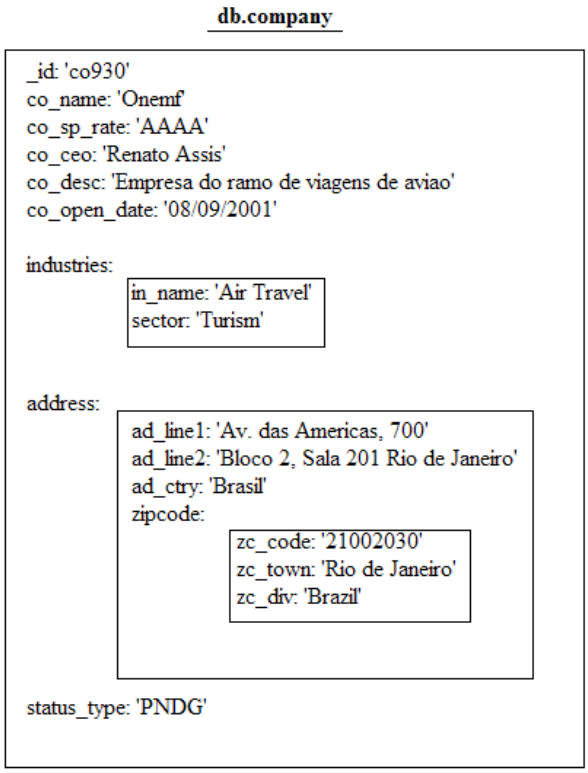

Figura 64: Exemplo de documento da coleção company.

Também criamos a coleção de *exchange* Figura 65. Esta representa os detalhes das corretoras como o nome, número de transações, data de abertura e fechamento, e o endereço. Nota-se que o endereço, assim como na coleção de empresa também é representado como um documento aninhado do documento principal da coleção *exchange*.

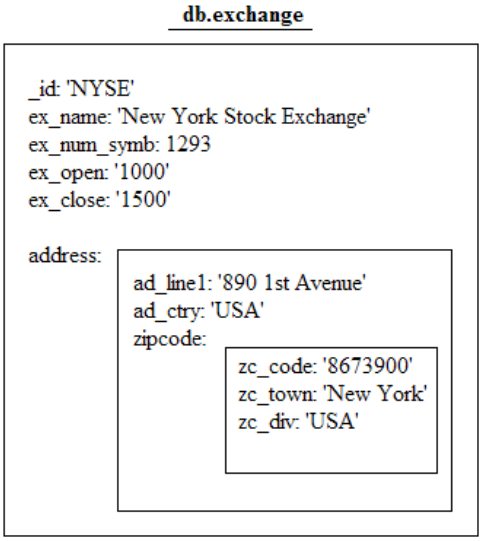

Figura 65: Exemplo de documento da coleção exchange.

Também foi criada uma coleção que controla as informações das transações (Figura 66). Esta coleção representa as informações de cada transação como nome, valores, datas de fechamento e abertura. Além disso, esta coleção também contém referências às coleções de *company* e *exchange* para relacionar a transação a uma determinada empresa e a uma determinada corretora.

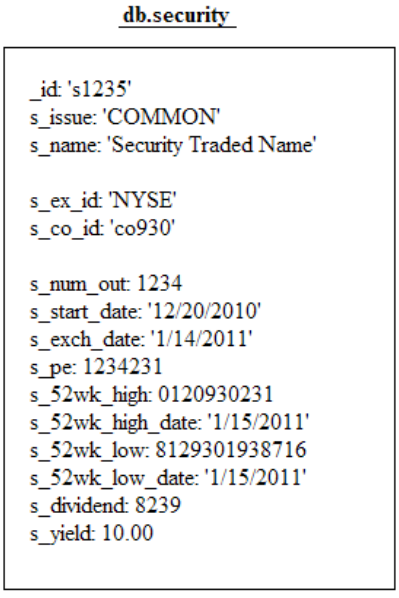

Figura 66: Exemplo de documento da coleção security.

3.5.2 Consultas

Selecionar o nome e sobrenome de todos os cliente cadastrados. Para retornar todos os nomes dos clientes do banco temos uma busca com um JSON vazio como filtro e o segundo parâmetro com os campos que devem ser retornados, que são os que possuem o primeiro e último nome de cada cliente.

db.customer.find({}, {'c_l_name' : 1, 'c_f_name': 1});

Selecionar todas as informações das *securities* da *watch list* cujo identificador é wl123. Para fazer a consulta que retorna as *securities* de uma determinada *watch_list*, necessitamos criar duas consultas ao nosso banco para obtermos o resultado, pois o MongoDB não permite junções. Essas duas consultas tiveram que ser tratadas por nós como mostrado na Figura 67: .

```
resultado = db.customer.find({'watch_list.wl_id' : wl123}, {'watch_list' : 1});
for res in resultado:
        for wl in res['watch_list']:
                if wl['wl_id'] == wl123:
                        res_final = db.security.find({'_id' : wl['id_security']})
```

Figura 67: Código utilizado feito em Python para retornar as securities de uma determinada watch_list

Selecionar todos os *brokers* que estão com status cancelado. Para realizar esta consulta, foi criado um filtro na consulta ao documento *broker*. Esse filtro seleciona apenas os documentos que possuem o campo b_st_id = 'CNCL', ou seja, o *broker* que possui o status cancelado.

db.broker.find({'b_st_id' : 'CNCL'});

Selecionar o setor em que atua a empresa (*company)* chamada Onemf. Para realizar a consulta que mostra em qual setor uma determinada empresa atua, criamos uma consulta na coleção *company* filtrando pelo nome da empresa.

db.company.find({'co_name': 'Onemf'}, {'industries.sector' : 1});

Selecionar todas as empresas em New York. Para descobrirmos quais empresas estão localizadas em New York, utilizamos a seguinte consulta na coleção *company*.

db.company.find({'address.zipcode.zc_town' : 'New York'});

Selecionar todos os clientes que possuem apenas uma conta associada. Para a consulta utilizada para descobrir quais clientes possuem apenas uma conta associada, utilizamos um modificador de consulta do MongoDB. Este modificador, chamado $size, permite saber qual o tamanho de um vetor especificado.

db.customer.find({'customer_accounts' : {$size : 1}});

Selecionar todas as empresas que foram abertas depois do ano de 2010. Para a consulta feita para descobrirmos quais empresas foram abertas após o ano de 2010, criamos um objeto do tipo data no JavaScript, permitindo assim a comparação com o modificador de consulta $gt (*greater than*) do MongoDB. Este modificador seleciona as empresas com o campo co_open_date maior que 1/1/2010.

db.company.find({'co_open_date' : {$gt : new Date(2010, 1, 1)}});

Selecionar todas as corretoras dos USA. Para a oitava consulta que buscava retornar as corretoras que estão localizadas nos EUA, não foi necessário aplicar junções como foi realizado no modelo relacional. Visto que o endereço é um documento inserido da coleção *exchange*, esse resultado foi obtido através da navegação nos documentos do MongoDB, como mostrado a seguir.

db.exchange.find({'address.ad_ctry' : 'USA'});

Selecionar todas as contas e permissões do cliente cujo identificador é c000. Para realizar a consulta que visa retornar todas as contas e permissões do cliente cujo identificador é o código c000 foi necessário um filtro na coleção *customer* em questão. Nesta consulta, são retornadas as customer_accounts que são sub documentos de cada cliente. Este sub documento possui o nome da conta, as permissões e outras informações.

db.customer.find({'_id' : c000}, {'customer_accounts' : 1 });

Selecionar a soma dos valores das taxas que cada cliente tem que pagar. Para descobrir qual o valor das taxas de cada cliente realizamos a consulta a seguir na coleção *customer*. Esta consulta não possui nenhum filtro e retorna o nome e sobrenome do cliente e o valor das taxas. Como as taxas são documentos inseridos dentro do próprio documento do *customer*, não foi necessária nenhuma junção.

Para obtermos o resultado da soma das taxas, é necessário um tratamento, percorrendo todos os documentos, ou mesmo a utilização do *map reduce*. No exemplo o tratamento foi feito pela aplicação e o código está na Figura 68.

resultado = db.customer.find({}, {'c_l_name' : 1, 'c_f_name': 1, 'taxrate.tx_rate' : 1});

```
for r in resultado:
        r['qntd_taxas'] = 0
        for taxa in r['taxrate']:
                r['qntd_taxas'] = r['qntd_taxas'] + taxa['tx_rate']
```

Figura 68: Exemplo de tratamento para a quantidade de taxas de cada usuário.

Selecionar o nome de todos os *brokers* que gerenciam mais de uma conta de cliente. A consulta para selecionar quais *brokers* gerenciam mais de uma conta não foi possível ser expressa através de consultas simples no MongoDB. Para conseguirmos esse resultado é necessária a utilização de uma função de *map reduce*, ou de um trecho de código representado na Figura 69. O exemplo copia o modelo de mapear o conjunto e então aplicar a seleção, e foi escrito na linguagem *Python*.

```
#MAP
resultado = db.customer.find({}, {'customer_account' : 1})
db.temp.drop()
for res in resultado:
        for ca in res['customer_account']:
                db.temp.insert({'broker_id': ca['broker_id']})

#REDUCE
resultado = db.temp.distinct('broker_id')
for res in resultado:
        if db.temp.find({'broker_id' : res}).count() >= 2:
                resultadoFinal = db.broker.find_one({'_id' : res})
```

Figura 69: Funções de map e reduce para a consulta de brokers com mais de uma conta associada.

Pode-se notar que as consultas feitas no MongoDB, quando possíveis, são bem simples de serem escritas, sendo expressas em poucas linhas.

3.6 REDIS

3.6.1 Estrutura

Para o nosso modelo, no Redis, foram criados 5 bases de dados. A primeira possui os detalhes das informações dos *Broker, Company, Customer, Customer Account, Exchange, Security* e *Taxrate*. Esta utiliza a estrutura de dados *hash*, onde existe uma chave que representa o identificador de cada um desses grupos que referenciam um *hash* com as informações deste grupo. Esse modelo está representado nas imagens que vão da Figura 70 até Figura 76.

No Redis não é obrigatória a crição da estrutura antes da inserção dos dados. Isso porque cada as chaves podem conter valores completamente distintos. Dessa maneira, o modelo é criado conforme os dados vão sendo inseridos. Os dados inseridos no Redis foram extraídos do modelo relacional através de consultas nas tabelas de customer, broker, company e assim por diante.

Os outros bancos de dados criados são dicionários de dados os quais foram criados de acordo com as consultas que queremos realizar. A base de dados 1 (Figura 77) implementa o dicionário de dados das *watch_list*. Neste dicionário existe uma lista de *securities* associada a cada *watch list* id. O banco de dados 2 (Figura 78) implementa uma lista de *watch list* associada a cada identificador do cliente. A base de dados 3 (Figura 79) implementa o dicionário de dados dos *brokers*, ou seja, neste dicionário existe uma lista de *customer account* associada a cada *broker id*. Finalmente, o banco de dados 4 (Figura 80) implementa uma lista de *customer account* associada a cada identificador de cliente.

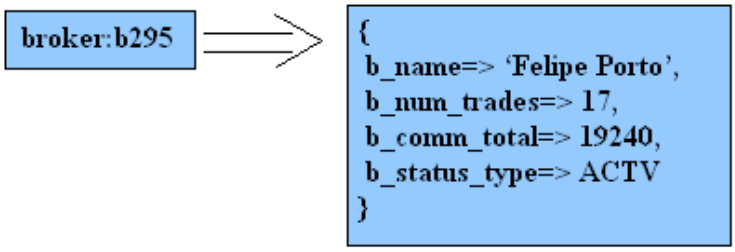

Figura 70: Banco 0 - Modelo do registro de Broker.

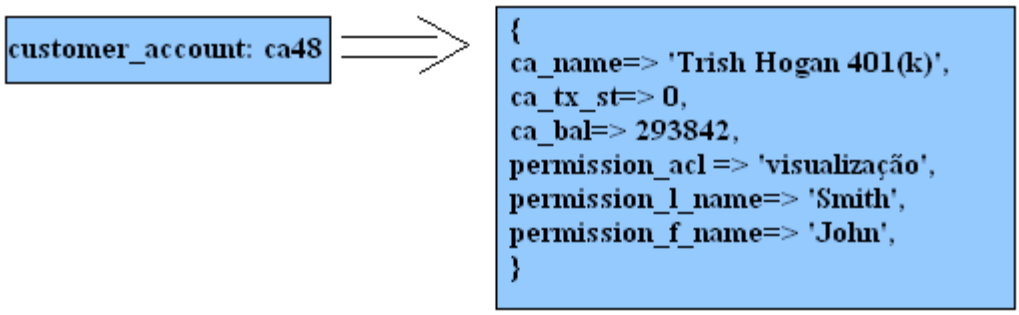

Figura 71: Banco 0 - Modelo do registro de Customer Account.

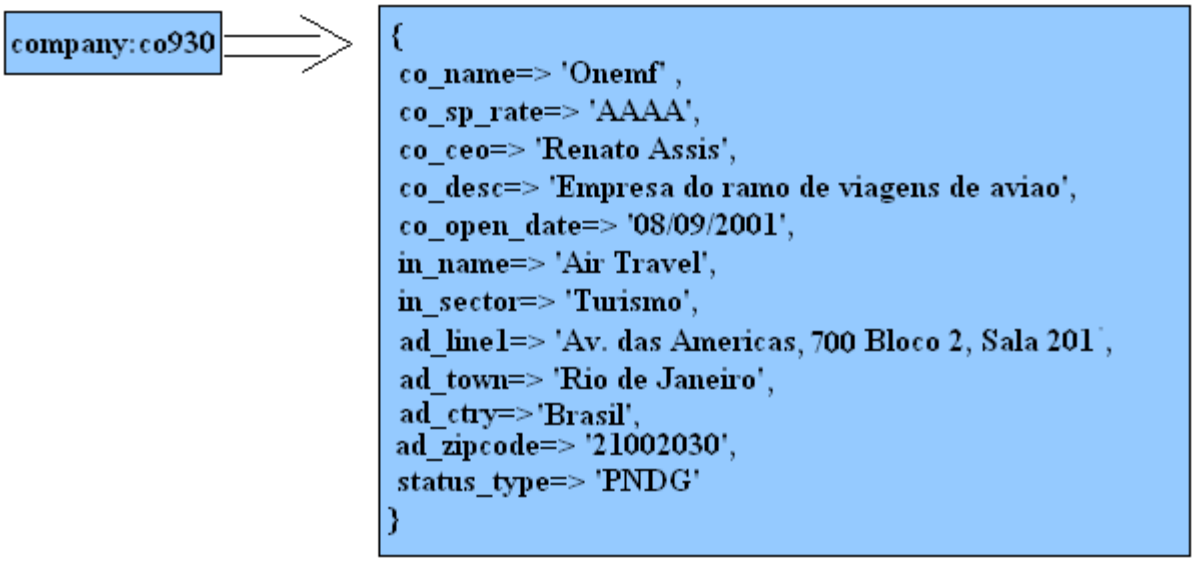

Figura 72: Banco 0 - Modelo do registro de Company.

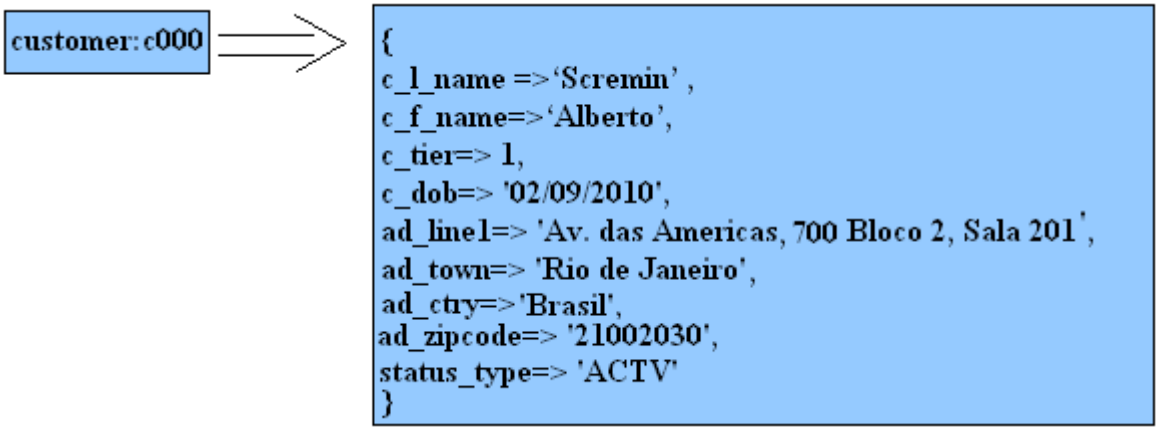

Figura 73: Banco 0 - Modelo do registro de Customer.

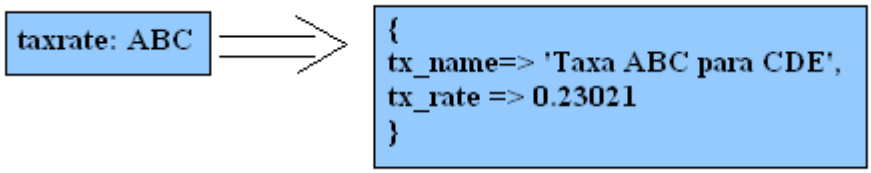

Figura 74: Banco 0 - Modelo do registro de Taxrate.

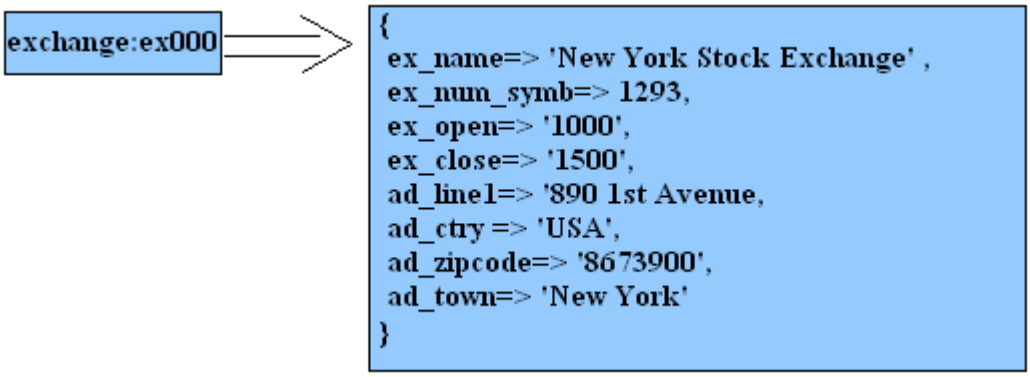

Figura 75: Banco 0 - Modelo do registro de Exchange.

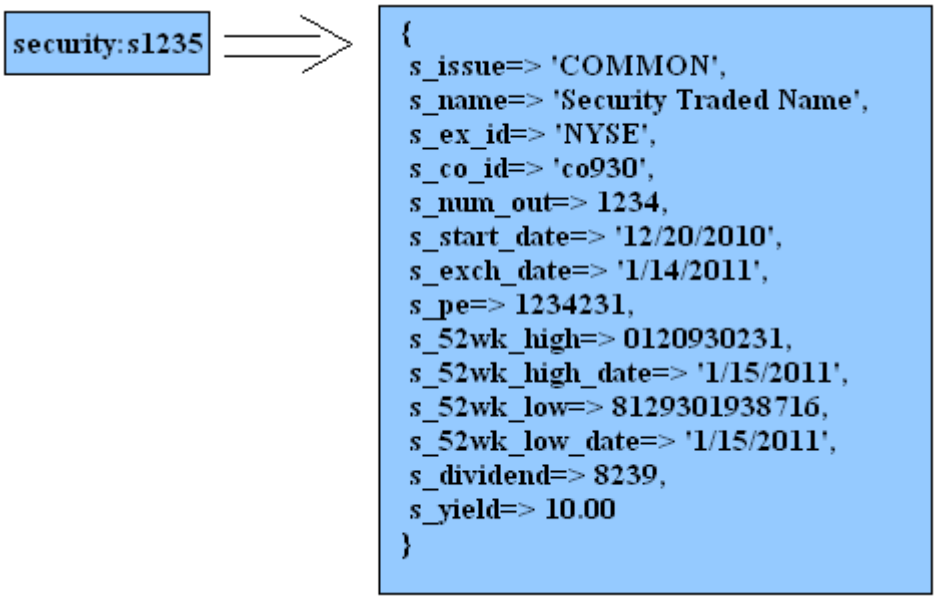

Figura 76: Banco 0 - Modelo do registro de Security.

No Redis, não é necessário utilizar um comando para criar um banco novo. Utiliza-se o comando *SELECT* para relizar a troca entre os bancos. Sendo assim, definimos quais dicionários ficariam em cada banco e utilizamos o comando *SELECT* <*numero*> para mudar de dicionário. Por

padrão o Redis pemite a utilização de 16 bancos de dados, entretanto, este número pode ser configurado.

Uma vez definidos os dicionários de dados, utilizamos o comando *LPUSH* para preencher as listas de cada chave. Por exemplo, para adicionarmos à chave wl123 as securities s1234 e s1635, utilizamos no banco de dados 1 o comando: *LPUSH* wl123 s1235 s1635. O mesmo serve para os outros dicionários de dados.

Estes dados foram obtidos através de consultas no MySQL nas tabelas que fazem o relacionamento. Por exemplo, na tabela *watch_item* está cadastrada a relação de *watch_list* com *security*, ou seja, sabemos quais *securities* estão associadas a cada *watch_list*. Desta maneira, conseguimos extrair as informações do dicionário de dados do banco de dados 1. Para criar a base de dados 2 utilizamos a tabela *customer_taxrate* e para as bases de dados 3 e 4 utilizamos a tabela *customer_account*, que possui a relação da conta com um determinado cliente que é gerenciada por um determinado *broker*.

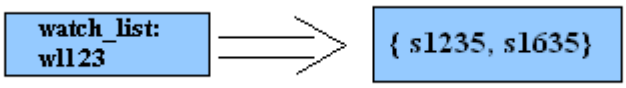

Figura 77: Banco 1 – Dicionário de dados das Watch Lists.

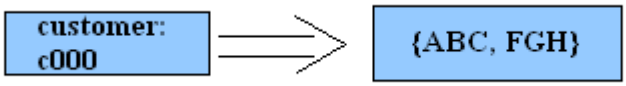

Figura 78: Banco 2 – Dicionário de dados das taxas dos clientes.

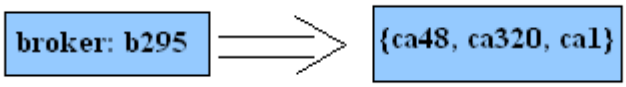

Figura 79: Banco 3 – Dicionário de dados das contas gerenciadas por cada broker.

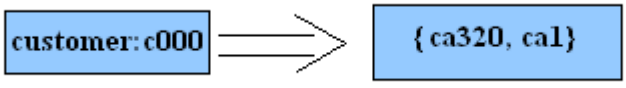

Figura 80: Banco 4 – Dicionário de dados das contas pertencentes a cada cliente

3.6.2 Consultas

No Redis algumas consultas envolveram a criação de um dicionário de dados e outras foram realizadas de uma maneira bem simples. Segue as consultas e uma breve explicação de cada comando.

Selecionar o nome e sobrenome de todos os cliente cadastrados. Na Figura 81, é utilizado o comando KEYS para listar todas as chaves dos cutomers. Em seguida é utilizado um laço de repetição, representado por um código *python*, para selecionar em cada chave de cliente o seu nome e sobrenome. Para isso, foi utilizado o comando *HMGET* que busca os campos c_l_name e c_f_name de determinada chave.

```
lista_customer_id = keys c*
for cust_id in lista_customer_id:
    hmget  cust_id c_l_name, c_f_name
```

Figura 81: Consulta do nome dos customers no Redis.

Selecionar todas as informações das *securities* da *watch list* cujo identificador é wl123. Na Figura 82, é realizada a consulta para trazer as informações das securities de cada *watch list*. Este foi um caso que precisamos utilizar o banco de dicionário de dados criado na base de dados 1, em que a estrutura de dados utilizada foi a lista. No banco de dados 1 possuímos uma lista de todos os identificadores das *securities* que pertencem a determinada *watch list*.

Assim, o comando de select muda qual o banco em que a consulta está sendo realizada. O comando *LRANGE* seleciona na chave wl123 a lista de identificadores. O primeiro valor é a posição de início e o último é a posição de fim, quando colocamos -1 na posição de fim, significa que estamos selecionando a lista inteira. Em seguida, de volta ao banco de dados 0, utilizamos um laço de repetição para trazer todas as as informações de cada chave da security, utilizando o comando *HGETALL*.

```
select 1
list_security_id = lrange wl123 0 -1
select 0
for security_id in list_security_id:
     hgetall security_id
```

Figura 82: Selecionar todas as informações das securities de uma determinada watch list no Redis.

Selecionar todos os *brokers* que estão com status cancelado. Na Figura 83, representamos a consulta que visa selecionar todos os *brokers* que estão com status cancelado. Para isso, utilizamos a base de dado 0, que possui as informações dos *brokers*. Utilizamos o comando *KEYS* para selecionar todos os *ids* de *brokers* existentes na base, e em seguida, um laço de repetição, para selecionar o campo status do *broker*. Caso ele possua o status cancelado, é selecionado, também com o comando *HGET*, o nome do determinado *broker*.

```
select 0
list_broker = keys b*
for broker_id in list_broker :
     broker_status = hget broker_id b_status_type
     if broker_status = 'CNCL' :
          hget broker_id b_name
```

Figura 83: Selecionar todos os brokers que estão com status cancelado, no Redis.

Selecionar o setor em que atua a empresa (*company)* chamada Onemf. A próxima consulta representada na Figura 84, tem por objetivo selecionar o setor de uma determinada empresa. Sendo assim, utilizamos o comando *KEYS* para selecionar todas as chaves de empresas cadastradas. Em seguida, para cada company_id, selecionamos o nome da empresa, utilizanod o comando *HGET*. Posteriormente, é feito um teste se a empresa em questão é a Onemf. Caso a igualdade seja verdadeira, selecionamos o setor desta empresa (in_sector) utilizando o comando *HGET*.

```
select 0
list_company = keys co*
for company_id in list_company :
    company_name = hget company_id co_name
    if co_name = 'Onemf' :
        hget company_id in_sector
```

Figura 84: Selecionar o setor de uma determinada company, no Redis.

Selecionar todas as empresas em New York. Na Figura 85, construímos a consulta que deve selecionar todas as empresas em New York. Para isso, utilizamos o comando *KEYS*, para selecionar todas as chaves de empresas cadastradas. Em seguida, para cada empresa, selecionamos o campo cidade do endereço e realizamos uma comparação. Se o campo for igual a New York, os dados empresa são mostrados.

```
select 0
list_company = keys co*
for company_id in list_company :
     company_town = hget company_id ad_town
     if company_town = 'New York' :
         hget company_id co_name
```

Figura 85: Selecionar, no Redis, todas as empresas em New York.

Selecionar todos os clientes que possuem apenas uma conta associada. Na consulta representada na Figura 86 desejamos selecionar todos os clientes que possuem apenas uma conta associada. Para isso, utilizamos o dicionário de dados do banco de dados 4 em que para cliente, possuímos uma lista de contas associadas.

Para trazer o resultado esperado, basta selecionar a quantidade de cada uma dessas listas, com o comando *LLEN*. Após esta seleção, verificamos se o valor é igual a 1. Caso seja, selecionamos no banco de dados 0 o nome e o sobrenome do cliente em questão.

```
select 4
lista_customer = keys c*
for customer_id in lista_customer :
    qtd_cust_account = llen customer_id
    if qtd_cust_account = 1 :
          select 0
          hmget customer_id c_l_name c_f_name
```

Figura 86: Selecionar, no Redis, todos os clientes que possuem apenas uma conta associada.

Selecionar todas as empresas que foram abertas depois do ano de 2010. Na Figura 87, construímos a consulta que deve selecionar todas as empresas que foram abertas após 2010. Para isso, utilizamos o comando *KEYS* para selecionar todas as chaves das empresas cadastradas. Em seguida, para cada empresa, selecionamos a data de abertura e realizamos uma comparação. Se o campo for maior ou igual a 01/01/2010, os dados da determinada empresa é mostrado.

```
select 0
list_company = keys co*
for company_id in list_company :
    company_date = hget company_id co_open_date
    if company_date >= '01/01/2010' :
        hget company_id co_name
```

Figura 87: Selecionar todas as companies que foram abertas depois do ano de 2010, no Redis.

Selecionar todas as corretoras dos USA. Na Figura 88, construímos a consulta que deve selecionar todas as corretoras dos USA. Para isso, utilizamos o comando *KEYS*, para selecionar todas as chaves de corretoras cadastradas. Em seguida, para cada *exchange*, selecionamos o país cadastrado e realizamos uma comparação. Se o campo for igual a USA, o nome da determinada transação é mostrada.

```
select 0
list_exchange = keys ex*
for exchange_id in list_exchange :
    exchange_ctry = hget exchange_id ad_ctry
    if exchange_ctry = 'USA' :
        hget exchange_id ex_name
```

Figura 88: Selecionar, no Redis, todos exchange dos USA.

Selecionar todas as contas e permissões do cliente cujo identificador é c000. Na próxima consulta representada na Figura 89, necessitamos das informações das contas do cliente c000. Inicialmente, utilizamos o comando *LLEN* para saber o comprimento da lista de contas do cliente c000, que está cadastrada no banco de dados 4. Desta maneira, ainda no banco de dados 4, criamos um laço de iteração para selecionar cada *customer account* que o mesmo possui. Utilizamos o comando *LINDEX* para selecionar esta chave e atribuimos o valor a uma variável chamada account_id. Finalmente, no banco de dados 0, selecionamos todas as informações desta conta e suas permissões, utilizando o comando *HGETALL* na chave que está guardada na variável account_id.

```
select 4
qtd_cust_account = llen c000
for posicao_ca in range(1, qtd_cust_account ) :
    select 4
    account_id = lindex c000 posicao_ca
    select 0
    hgetall account_id
```

Figura 89: Selecionar todas as contas e permissões de um dado cliente, no Redis.

Selecionar a soma dos valores das taxas que cada cliente tem que pagar. Na Figura 90 utilizamos outro dicionário de dados, que também utiliza a estrutura de dados lista onde cada chave de cliente

possui associada a ela uma lista de id das taxas que este cliente possui, utilizando o banco de dados 2. Assim, para realizar a consulta de selecionar o somatório das taxas que cada cliente possui, deve-se selecionar o id da customer_account deste cliente, utilizando o comando *LINDEX*. Em seguida, devemos buscar o valor desta conta, com o comando *HGET*, no banco que possui as informações das taxas, o banco de dados 0. Caso o cliente possua mais de uma conta, devem-se acumular os valores das taxas.

Como já foi mencionado, o Redis não permite uma busca sobre toda a base, somente buscas por chaves. Dessa maneira, precisamos criar uma iteração na aplicação para buscar todas as chaves de clientes existentes, representada pelo código *python* na Figura 90.

```
select 2
list_customer = keys c*
for customer_id in list_customer :
    select 2
    qtd_taxas = llen customer_id
    for posicao_ca in range (1, qtd_taxas):
        c_ca_id =  lindex customer_id posicao_ca
        select 0
        custa_tx = hget c_ca_id tx_rate
        tot_taxas = tot_taxas + custa_tx
    print customer_id, 'possui', tot_taxas
```

Figura 90: Selecionar, no Redis, a quantidade de taxas cada cliente possui.

Selecionar o nome de todos os *brokers* que gerenciam mais de uma conta de cliente. A próxima consulta, representada na Figura 91, não pode ser realizada por completo através do Redis. Desta forma, é necessária uma intervenção da aplicação. Utilizamos mais um dicionário de dados, criado na base de dados 3. Este dicionário possui a lista de contas de clientes que cada *broker* gerencia.

Sendo assim, selecionamos para cada identificador de *brokers* a quantidade de contas que são gerenciadas, através do comando *LLEN*. Desta maneira, é necessário realizar um teste na aplicação para verificar se este valor é maior do que 1. Caso seja, é executada a segunda parte da consulta, onde é selecionado o campo b_name deste identificador através do comando *HGET*, no banco de dados 0.

```
select 3
list_broker = keys b*
for broker_id in list_broker:
   qtd_broker = llen broker_id
   if qtd_broker > 1 :
       select 0
       hget broker_id b_name
```

Figura 91: Selecionar, no Redis, o nome dos brokers que gerenciam mais de uma customer_account.

3.7 CONSIDERAÇÕES FINAIS

Neste capítulo mostramos como um modelo relacional de uma aplicação pode ser representado nos outros SGBDs não relacionais. Além disso, também foi apresentada uma série de consultas que podem ser realizadas em cima deste modelo. Representamos estas consultas no modelo relacional e mostramos como as mesmas deveriam ser representadas no modelo não relacional.

Na Tabela 1, realizamos uma comparação entre cada consulta nos diferentes SGBDs. Os campos que estão preenchidos com a palavra SIM indicam que as consultas puderam ser realizadas inteiramente pelo SGBD. Os campos que estão preenchidos com a palavra APLC indicam que foi necessário escrever um código externo à consulta para que fosse possível recuperar todos os dados. Vale notar que não houve nenhum caso em que as consultas não puderam ser realizadas pelos SGBDs em questão.

CONSULTA / SGBD	1	2	3	4	5	6	7	8	9	10	11
MySQL	SIM	SIM	SIM	SIM	SIM	SIM	SIM	SIM	SIM	SIM	SIM
Sedna	SIM	SIM	SIM	SIM	SIM	SIM	SIM	SIM	SIM	SIM	SIM
Cassandra	APLC	APLC	SIM	SIM	SIM	APLC	SIM	SIM	APLC	APLC	APLC
MongoDB	SIM	APLC	SIM	SIM	SIM	SIM	SIM	SIM	SIM	APLC	APLC
Redis	APLC	APLC	APLC	APLC	APLC	APLC	APLC	APLC	APLC	APLC	APLC

Tabela 1: Tabela comparativa das consultas realizadas.

4 CONCLUSÃO

O MySQL possui uma estrutura rígida, onde toda e qualquer tupla necessita possuir todas as colunas da tabela. Entretanto, apesar desta rigidez, ele nos permitiu realizar todas as consultas sem nenhuma dificuldade. O mesmo nos permitiu realizar desde as consultas mais simples até as consultas mais complexas, utilizando operadores de comparação, agregação e junções.

O MongoDB não requer que seja criado o esquema antes que os documentos sejam inseridos, dessa forma facilmente permite começarmos o nosso banco. Porém, não é recomendado que todos os tipos de documentos fiquem em uma mesma coleção, pois isto pode onerar o desempenho das consultas. Desta maneira, devemos criar um esquema para cada conjunto de documentos que compartilham a mesma coleção.

Na questão de como fazer as consultas no MongoDB, apesar da maioria das consultas serem expressas facilmente, outras não puderam ser expressas sem o auxílio de um tratamento da aplicação do usuário, aumentando assim o esforço necessário para trabalhar com tal SGBD.

O mesmo modelo que foi utilizado no MongoDB, também foi utilizado para o Sedna. O Sedna utiliza de um modelo concebido em XML Schema. O XML Schema, permite uma grande flexibilidade para a construção de seus modelos e, dependendo da experiência do usuário, pode acabar apresentando uma linguagem mais complexa para sua construção. Para as consultas no Sedna, utilizamos o padrão XQuery, que se mostrou como uma ferramenta muito poderosa para a tarefa, permitindo expressar todas as consultas feitas no MySQL, mesmo tendo um esquema diferente do MySQL. Este padrão de linguagem permitiu a criação desde as consultas mais simples às mais complexas.

O Redis, assim como o MongoDB, permite a criação de um modelo de dados sem uma estrutura rígida, ou seja, os campos podem ser adicionados conforme a necessidade. Devido ao modelo e as consultas propostas neste trabalho, foi necessária a criação de 5 bancos no Redis. O primeiro para as informações de cada chave, armazenada em um *hash* e os outros trabalhando como dicionário de dados, onde cada chave está relacionada a uma lista de valores. O Redis não possui nenhum mecanismo para manter a integridade referencial. Sendo assim, se um dado for atualizado na primeira base de dados, a aplicação deve cuidar de atualizar os dicionários de dados. Consequentemente, essa é mais uma complexidade que é abstraída do SGBD e inserida na aplicação.

Outro fator dificultante do Redis foi a consulta, pois o mesmo só realiza consultas através das chaves, então as consultas sobre toda a base de dados não puderam ser realizadas completamente no Redis. Para resolver este problema, foi necessário criar um laço de repetição na aplicação, onde em cada iteração seria selecionado o campo de uma determinada chave até o momento em que todas as chaves tenham sido lidas. As consultas ao Redis ficaram bem simples, entretanto, algumas ganharam um nível de dificuldade devido às limitações do Redis.

O Cassandra, assim como o Redis, é um banco de dados orientado a chave-valor, porém apesar de possuírem o mesmo princípio de estrutura, eles são bastante diferentes. O modelo de colunas implementado no Cassandra permite que para cada chave, existam atributos, o que no Redis é tratado com uma estrutura de dados *hash*.

No Cassandra, encontramos muitos fatores limitantes, como a ausência de algumas estruturas de dados, como a lista, os operadores de repetição e agregação. Com isso, todas as consultas que

realizamos precisaram de um tratamento especial na aplicação, ou seja, as consultas realizadas no banco se tornaram mais simples, porém o código da aplicação mais complexo. Assim como no Redis, a complexidade é abstraída do SGBD e inserida na aplicação.

Como trabalhos futuros, poderíamos adicionar outros bancos não relacionais a este estudo. Também poderia ser utilizado o esquema completo do TPC como trabalho futuro, além de utilizar o desempenho, a replicação, o *backup* e o *restore* como outros critérios comparativos entre os SGBDs.

5 REFERÊNCIAS BIBLIOGRÁFICAS

ANSI/ISO. SQL-86, 1986.

BEAULIEU, A. Learning SQL. 2nd ed. Beijing; Sebastopol: O'Reilly, 2009.

BOAG, S.; CHAMBERLIN, D.; FERNANDEZ, M. F.; et al. XQuery 1.0: An XML Query Language. Recuperado maio 17, 2011, de http://www.w3.org/TR/xquery/, 2010.

BOGONI, L. P. SQL Magazine. Por dentro do movimento NoSQL, n. 83, 2010.

BRAY, T.; PAOLI, J.; SPERBERG-MCQUEEN, C. M.; MALER, E.; YERGEAU, F. Extensible Markup Language (XML) 1.0 (Fifth Edition). Recuperado maio 24, 2011, de http://www.w3.org/TR/REC-xml/, 2008.

BUNEMAN, P. Semistructured data. ACM SIGACT-SIGMOD-SIGART, PODS '97, p. 117–121, 1997.

CHODROW, K.; DIROLF, M. MongoDB: The Definitive Guide. 1st ed. O'Reilly Media, 2010.

CHODROW, K.; GILL, M. Querying - MongoDB. Recuperado maio 12, 2011, de http://www.mongodb.org/display/DOCS/Querying, 2010.

CLARK, J.; DEROSE, S. XML Path Language (XPath). Recuperado junho 9, 2011, de http://www.w3.org/TR/xpath/, 2003.

CODD, E. F. A Relational Model of Data for Large Shared Data Banks,1970.

CUONG, N. XML Native Database Systems Review of Sedna, Ozone, NeoCoreXMS ,2006.

DEAN, J.; SANJAY, G. MapReduce: Simplified Data Processing on Large Clusters, v. Communications of the ACM 51(1): 107-113, 2008, 2008.

DUBOIS, P. MySQL. Indianapolis IN: New Riders Pub. 2000.

DUBOIS, P. MySQL 5.0 Certification Study Guide. 1st ed. Indianapolis Ind: MySQL Press, 2008.

Ecma International. JSON: The Fat-Free Alternative to XML. Recuperado maio 12, 2011, de http://www.json.org/xml.html, 2006.

ELLIS, J. Cassandra Developer Center - What's new in Cassandra 0.7: Secondary indexes. Recuperado maio 17, 2011, de http://www.datastax.com/dev/blog/whats-new-cassandra-07-secondary-indexes, 2010, março 12.

ELMASRI, R.; NAVATHE, S. Sistemas de banco de dados. 4th ed. São Paulo: Pearson Addison Wesley, 2009.

FINDLEY, R. Redis Overview - SlideShare. Recuperado maio 19, 2011, de http://www.slideshare.net/neomindryan/redis-overview-presentation, 2010.

GRINEV, M.; FORMICHEV, A.; KUZNETSOV, S. Sedna: A Native XML DBMS,2004.
HEWITT, E. Cassandra: the definitive guide. Beijing: O'Reilly, 2011.

HOROWITZ, E.; STEARN, M. MapReduce - MongoDB. Recuperado maio 17, 2011, de http://www.mongodb.org/display/DOCS/MapReduce, 2011.

Institute for System Programming RAS. Sedna Programmer's Guide, 2003.
MERRIMAN, D. Query Optimizer - MongoDB. Recuperado maio 13, 2011, de http://www.mongodb.org/display/DOCS/Query+Optimizer, 2010.

MERRIMAN, D.; CHODROW, K. Database References - MongoDB. Recuperado maio 16, 2011, de http://www.mongodb.org/display/DOCS/Database+References, 2011.

MURPHY, R.; MERRIMAN, D. Indexes - MongoDB. Recuperado maio 12, 2011, de http://www.mongodb.org/display/DOCS/Indexes, 2011.

MURPHY, R.; VOYER-PERRAULT, G. Schema Design - MongoDB. Recuperado maio 9, 2011, de http://www.mongodb.org/display/DOCS/Schema+Design, 2011.

RUSSO, M. J. Redis, from the Ground Up. Recuperado maio 23, 2011, de http://blog.mjrusso.com/2010/10/17/redis-from-the-ground-up.html, 2010.

SANFILIPPO, S.; NOORDHUIS, P. Command reference – Redis. Recuperado maio 19, 2011, de http://redis.io/commands, 2010.

STEPHENS, R. Beginning Database Design Solutions. Indianapolis: Wiley Pub, 2009.

STEPPAT, N. Bancos de dados não relacionais e o movimento NoSQL | blog.caelum.com.br. Recuperado junho 29, 2011, de http://blog.caelum.com.br/bancos-de-dados-nao-relacionais-e-o-movimento-nosql/, 2009.

W3C XML Schema. Recuperado junho 8, 2011, de http://www.w3.org/XML/Schema.