Departamento de Ciências e Tecnologia da Informação

Tradutor de Interrogações SQL para MongoDB

Vicente Germano Pereira

Dissertação submetida como requisito parcial para obtenção do grau de
Mestre em Engenharia de Telecomunicações e Informática

Supervisor:
Dr. Pedro Ramos, Professor Associado
ISCTE-IUL

Setembro 2018

Agradecimentos

Em primeiro lugar, quero agradecer ao meu orientador, Professor Doutor Pedro Ramos, por todo o apoio prestado, pela orientação fundamentais no desenvolvimento da dissertação e, principalmente, pela disponibilidade total em ajudar, sempre que necessário.

Quero agradecer à minha família e à minha namorada, por se mostrarem preocupados e prontos a ajudar, e, especialmente, por sempre acreditarem em mim e nas minhas capacidades.

Por último, mas não menos importante, quero agradecer a todos os meus colegas, que passaram muitas e muitas horas comigo no ISCTE-IUL. Pela companhia, pela entreajuda e por partilharem comigo os maus e os bons momentos ao longo do último semestre.

Resumo

Nos últimos anos, com o crescimento exponencial dos dados semiestruturados e não estruturados, e com o aumento massivo do seu volume, as bases de dados relacionais que não foram projetadas para lidar com estes desafios de escalabilidade e agilidade que os aplicativos modernos enfrentam, nem foram construídos para aproveitar o poder de armazenamento e processamento, atualmente disponível, forçaram as organizações a desenvolver novos sistemas, surgindo, então, as bases de dados NoSQL, incluindo MongoDB.

Sendo uma tecnologia relativamente recente, a maior parte dos utilizadores desconhecem a sua estrutura e linguagem de interrogação. Se existisse uma ferramenta que possibilitasse, não só, os utilizadores interrogarem uma base de dados MongoDB, através de comandos SQL, nomeadamente comandos *select*, baseando-se num modelo relacional, do qual já estão habituados, mas também aprenderem a linguagem de interrogação Mongo, com a sua utilização, então, esta nova tecnologia tornar-se-á vantajosa para um maior número de utilizadores.

Assim sendo, nesta dissertação foi desenvolvida uma ferramenta que, primeiro, transpõe o modelo de coleções MongoDB para o modelo relacional correspondente. Permitindo aos utilizadores, a partir do modelo previamente estabelecido, aceder à informação armazenada na base de dados, com uma interrogação *select*, que será automaticamente traduzida para a interrogação mongo equivalente. O utilizador terá, ainda, a possibilidade de conhecer qual a interrogação mongo equivalente, dando-lhe a oportunidade de aprender a linguagem de interrogação desta base de dados.

Palavras Chave: Estrutura Aggregate, Find, Modelo Relacional, MongoDB, SQL

Abstract

In the last years, with the exponential growth of semi-structured and unstructured data, and with the massive increase in its volume, relational databases were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of storage and processing power available today, forced organizations to develop new systems, emerging then the NoSQL databases, including MongoDB.

Being a relatively recent technology, most users are unaware of its structure and query language. If there was a tool that would enable, not only, users querying a MongoDB database, through SQL commands, namely select queries, based on a relational model of which they are already accustomed, but also learn the mongo query language, with its use, then this new technology will become advantageous for a greater number of users.

Thus, in this dissertation a tool was developed that first convert the MongoDB collections model to the corresponding relational model, allowing users, from the previously established model, to access the data stored in the database, with a select command, which will be automatically translated for the equivalent mongo query. The user will also be able to know which mongo equivalent query, giving him the opportunity to learn the query language of this database.

Keywords: Aggregate Framework, Find, MongoDB, Relational Model, SQL

Índice

Índice de Figuras

Índice de Tabelas

Acrónimos

ACID – Atomicidade, Consistência, Isolamento, Durabilidade

ADS – *Advertising*

API – *Application Programming Interface*

ASC - Ascendente

BSON – *Binary JSON*

CAP – Consistency, Availability, *Partiton*

CSV – *Comma-separated values*

DESC - Descendente

GB – *GigaByte*

GUI – *Grafic User Interface*

HTML – *HyperText Markup Languag*

ID – Identidade Digital

JDBC – *Java Database Connectivity*

JSON – *JavaScript Object Notation*

KB – *Kilobyte*

NoSQL – *Not Only SQL*

ODBC – *Open Database Connectivity*

PODC – *Principles of Distributed Computing*

RAM – *Random Access Memory*

RDBMS – *Relational Database Management System*

SQL – *Structured Query Language*

URL – *Uniform Resource Locator*

XML – *eXtensible Markup Language*

Capítulo 1 – Introdução

1.1 Enquadramento

Durante muito tempo, a informação tem sido armazenada em bases de dados relacionais, formadas por tabelas, constituídas por colunas e linhas de maneira estruturada e consistente, tanto no tamanho, como no tipo ou em outras restrições. Com a quantidade crescente de dados, surgiram grandes aplicações escaláveis e as organizações sentiram necessidade de mudar, encontrando valor na rápida análise e captura de grandes quantidades de dados, encontrando solução nas bases dados NoSQL.

Estas bases de dados estão a ganhar visibilidade e a ser utilizadas em cada vez mais domínios. A sua linguagem de interrogação é relativamente desconhecida (face, por exemplo, à linguagem relacional), o que impossibilita a sua utilização por utilizadores não profissionais que não tenham acesso a uma ferramenta de conversão de fácil utilização. A linguagem SQL (nomeadamente o comando *select*) é acessível e utilizada por uma vasta comunidade de utilizadores. Se um utilizador puder interrogar uma base de dados não relacional através de comandos SQL, então, esta nova tecnologia passará a estar disponível a um número muito maior de utilizadores.

1.2 Objetivos

Enfrentando este problema, o objetivo da dissertação é desenvolver um mecanismo de tradução automática de interrogações SQL para interrogações a uma base de dados MongoDB. O mecanismo de tradução pressupõe o prévio estabelecimento de um conjunto de regras de mapeamento entre o modelo relacional e o modelo de coleções JSON. Se as regras permitirem um mapeamento bidirecional, a partir de um modelo JSON é possível encontrar um modelo relacional correspondente. Após as regras estarem estabelecidas e o modelo relacional ser apresentado ao utilizador, este não necessita conhecer a base de dados MongoDB para a interrogar através da utilização da linguagem SQL.

Assim sendo, os objetivos que se colocam na presente dissertação são:

1. Definir um conjunto de regras de mapeamento entre o modelo de coleções, MongoDB, e o modelo relacional, sem que haja perda de informação;

2. Converter as interrogações SQL pretendidas em interrogações MongoDB equivalentes, de modo a que a obtenção dos dados seja idêntica em ambos os casos.

De forma a testar a solução encontrada efetuou-se um conjunto de testes que validem o trabalho efetuado. A aplicação foi desenvolvida na linguagem Python.

1.3 Contribuições

Foi desenvolvida uma ferramenta que permite ao utilizador, não só, aceder a dados numa base de dados MongoDB, como a utiliza-la no âmbito da educação/ensino.

Artigo de revista submetido no jornal internacional, *Future Generation Computer Systems*. (ISSN: 0167-739), ed Elsevier Editorial System.

1.4 Estrutura e organização da dissertação

O seguinte estudo encontra-se estruturado em sete capítulos, da seguinte forma.

O primeiro capítulo introduz os objetivos, motivações e contribuições do tradutor desenvolvido.

O segundo capítulo reflete uma revisão geral teórico incluindo o enquadramento do leitor na temática.

O terceiro capítulo refere-se ao, essencial, estudo detalhado da linguagem de interrogação Mongo

O quarto capítulo apresenta as regras de transposição do modelo de coleções Mongo para o modelo relacional;

O quinto capítulo apresenta o sistema de tradução de interrogações *select*, para as interrogações Mongo equivalentes;

O sexto capítulo explica como o utilizador deve proceder com a ferramenta, juntamente com a estrutura do código de como foi desenvolvida;

O sétimo capítulo refere-se às conclusões finais sobre o tradutor desenvolvido, juntamente com melhorias que possam ser feitas, no futuro.

Capítulo 2 – Revisão da Literatura

2.1 NoSQL

O termo NoSQL apareceu em 1998, contudo, só em 2009, após Joham Oskarsson organizar um encontro para pessoas interessadas em armazenamento de dados estruturados e distribuídos, e Eric Vans usá-lo para citar o aumento das bases de dados não relacionais, é que este ressurgiu. "NoSQL" ou também chamado "*Not Only* SQL" não é uma base de dados, nem sequer um tipo de base de dados, é um termo utilizado para representar todas as bases de dados que não são relacionais, e que por isso, não utilizam a linguagem SQL para criar as suas interrogações (Vaish 2013).

Muitas organizações sentiram necessidade de mudar, desenvolvendo os seus próprios sistemas NoSQL. Estas bases de dados vêm combater problemas como a escalabilidade, disponibilidade e performance. Nas próximas secções, serão estudados estes sistemas, demonstrando as suas principais características, e no que elas diferem perante as bases de dados tradicionais.

2.1.1 Livre de Esquema

No mundo relacional, antes de inserirmos quaisquer registos, é necessário desenhar a estrutura da base de dados, isto é, definir tabelas, colunas, e os tipos de dados do qual vai ser constituída, de modo a que cumpra as regras de normalização. A estrutura da base de dados definida, precisa de ser cumprida, só sendo permitido introduzir dados que vão de encontro com essa mesma estrutura. Além do mais, com a utilização destes sistemas, quando os dados não "encaixam" facilmente em tabelas, a estrutura da base de dados pode ficar bastante complexa, difícil e lenta para trabalhar (Leavitt 2010).

Isto tornasse uma clara desvantagem perante as bases de dados NoSQL, onde não é necessária uma estrutura pré-definida, estando livre de adicionar ou agregar qualquer tipo de dados, sem conhecimento prévio de como a base de dados guarda a informação. Por esta razão, as bases de dados NoSQL, são uma melhor escolha para sistemas que necessitam de lidar com dados não estruturados, como e-mails, processamento de texto ou multimédia (Fowler 2015).

2.1.2 Agregação

Numa base de dados relacional, a informação é estruturada em diferentes tabelas, que se relacionam entre si, de forma a evitar ao máximo informação redundante. Considerando uma base de dados de uma empresa, onde um empregado pode ter várias moradas e vários contactos, é de esperar que o sistema armazene a informação sobre os contactos (*contact*) e as moradas (*Address*) de todos os empregados (*Employee*), Figura 2-1.

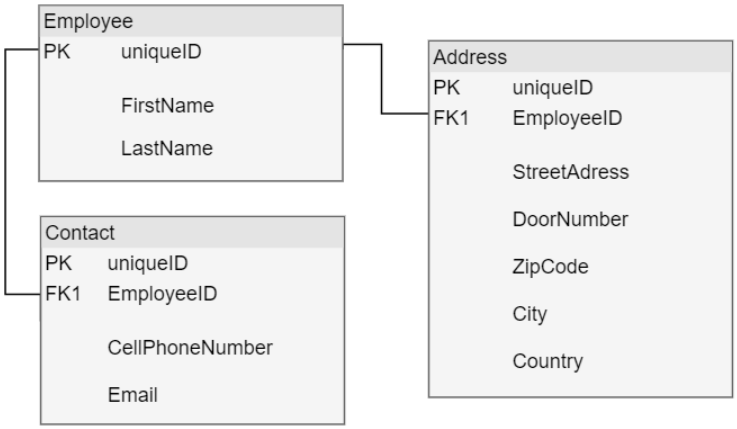

Figura 2-1 Exemplo de uma base de dados, de uma empresa, que guarda a informação sobre os empregados

Supondo que é necessário consultar a informação sobre um certo empregado, incluindo os seu contactos e cidade onde mora, seria necessário recorrer a um *join*, que contenha as chaves estrangeiras das tabelas intervenientes, como ilustrado na Figura 2-2 (na figura utiliza-se o operador *join* embora a sua utilização não seja obrigatória).

```
SELECT e.FirstName, e.LastName, c.CellPhoneNumber, c.Email, a.City
FROM Employee e
JOIN Contact c ON c.EmployeeID = e.uniqueID
JOIN Address a ON a.EmployeeID = e.uniqueID
```

Figura 2-2 Interrogação utilizada para consultar a informação sobre um empregado

Uma base de dados NoSQL não tem este tipo de relações, isto significa que, em vez de guardar os dados em múltiplas tabelas, toda a informação do empregado poderá ser agregada e armazenada num único registo, Figura 2-3.

```
  {
 "id": "1",
 "firstName": "João",
 "lastName": "Silva",
 "addresses": [
    {
       "streetAdrees": "Avenida das Forças Armadas",
       "doorNumber": 10,
       "zipCode": "1234-567",
       "city": "Lisboa",
       "country": "Portugal"
    }
                ],
 "contact": [
    {"email": "joao.silva@xpto.com" },
    {"cellPhoneNumber": "+351 912345678"}
 ]
}
```

Figura 2-3 Informação sobre o empregado agregado num único documento

Esta abordagem, onde os dados estão desnormalizados, caracteriza-se por ser mais fácil de guardar os dados e mais simples e rápido de fazer as consultas. Em situações simples, como a anterior, se fosse necessário alterar a informação sobre um empregado, numa base dados tradicional, seria necessário operar sobre várias tabelas, ao contrário de no exemplo da Figura 2-3, onde apenas seria necessário operar sobre um documento. Além disso, numa base de dados relacional uma interrogação pode relacionar várias tabelas, tabelas que quanto mais restrições tiverem mais lento o sistema tornam.

2.1.3 Escalável Horizontalmente

Ao contrário das bases de dados tradicionais, os sistemas NoSQL têm um escalonamento horizontal. Isto significa que, em vez de existir um único servidor, onde é armazenada toda a informação, vão existir muitos mais, pequenos e simples, servidores interligados, chamados clusters, onde os dados vão ser repartidos e replicados ao longo da rede.

Antigamente, com o aumento das quantidades de informação, era necessário comprar processadores mais rápidos e caros, porém a densidade de dados era tanta, que mesmo estes processadores poderiam causar um atraso na resposta do sistema. Este fenómeno levou as organizações a mudar os seus sistemas, focando-se em aumentar a performance, não comprando melhores processadores, mas sim comprando vários processadores mais baratos e simples, que dividiam as tarefas entre si. Quantos mais processadores maior quantidade de dados o sistema conseguia lidar (McCreary and Kelly 2014).

A utilização destes clusters, tornou o sistema bastante mais rápido, tanto a ler como a escrever dados. Por outro lado, os dados agora replicados e particionados ao longo da rede, trouxeram ao sistema uma grande disponibilidade e durabilidade, isto porque, mesmo que um dos nós falhe, continuaremos a receber a resposta, porque o nó estragado poderá ser reposto por outro (Hecht and Jablonski 2011).

2.2.4 Partição

Com a utilização de vários subsistemas interligados e a necessidade de distribuir os dados, pelos mesmos, surgiram duas abordagens para fazer partição dos dados. Desde que o modelo de dados, da maior parte dos sistemas NoSQL, é orientado a chaves, segundo (Hecht and Jablonski 2011), existem duas estratégias de partição:

Partição Range

É caracterizada por existir um servidor de encaminhamento designado, que divide o conjunto de chaves em vários blocos, e aloca-os em diferentes nós. Como o servidor de encaminhamento é o responsável pela gestão das chaves, a disponibilidade de todo o cluster depende do mesmo. Para encontrar uma certa chave, o cliente deve comunicar com o servidor de encaminhamento, requerendo a tabela de partições. Os dados são guardados com base no

intervalo da chave, ou seja, se pensarmos nas chaves associadas á idade de uma pessoa, é de esperar que estas estejam dividias em partições de intervalos de idade até aos 30 anos, dos 30 aos 60, e dos 60 aos 90, por exemplo. Esta estratégia tem a vantagem de tratar eficientemente interrogações que consultam campos com valores que podem ser representados num intervalo, chamadas "r*ange queries".* Listar todos os empregados com idade entre 20 e os 30 anos é um exemplo de uma *range querie*.

Partição Hash

Em contraste com a estratégia anterior, as chaves são distribuídas por funções de *hash*. Não existe apenas um servidor que gere as chaves, mas cada servidor é responsável por um certo intervalo de *hash*. As operações dentro do mesmo cluster, nesta estratégia, podem ser concluídas rapidamente. Além do mais, ao contrário da primeira estratégia, adicionar e remover nós, apenas afeta um pequeno subconjunto de todo o cluster. No entanto, se for necessário operar *range queries,* o desempenho pode diminuir pois as chaves do mesmo intervalo estão distribuídas ao longo do cluster.

2.1.5 Base vs Acid

Imaginem agora que uma aplicação está dividida em múltiplos servidores ao longo do país, onde cada servidor está responsável por uma certa função. Se um dos servidores, por algum motivo, falhar, o que devemos fazer? Devemos fechar a aplicação e dizer aos utilizadores que voltem mais tarde, atrasando o sistema? Com intuito de aumentar a performance e a disponibilidade do sistema, as bases de dados NoSQL permitem uma certa inconsistência nos dados, ou seja, responder às interrogações mesmo sem ter a certeza que a informação ainda está devidamente atualizada. Esta abordagem tem o nome de BASE:

BASE – *Basically available*, *Soft state*, *Eventually consistent*.

A ideia de prescindir das restrições *ACID* do modelo relacional (onde a consistência permanente é central), devesse ao facto, dos sistemas *BASE* darem prioridade à disponibilidade em detrimento da consistência. Mesmo que haja alguma inconsistência nos dados, o essencial é manter o sistema disponível e com alta performance durante todo o tempo. A disponibilidade nos sistemas *BASE*, é alcançada pela capacidade do sistema suportar falhas na rede (Pritchett 2008).

Contudo, se é obrigatório existir consistência nos dados, como por exemplo, em sistemas de finanças, onde é necessário fazer a gestão de transações que envolvem dinheiro, as bases de dados relacionais são preferenciais. Isto não quer dizer, que os sistemas NoSQL no possam cumprir estas restrições. Atualmente, muitas bases de dados oferecem diferentes configurações entre a consistência e a disponibilidade, consoante as necessidades de cada um (Hecht and Jablonski 2011).

2.1.6 CAP

Eric Brewer, no PODC 2000 (Eric Brewer 2000), desenvolveu a seguinte conjuntura: um sistema distribuído não pode proporcionar estas três características, de uma só vez:

- **Consistência**: para um sistema distribuído garantir consistência, significa que todos os nós, no mesmo espaço de tempo, devolvem o mesmo valor, o valor mais recente. Segundo (Gilbert and Lynch 2005), o melhor modo de explicar a consistência é a partir de um objeto atómico. Por exemplo, uma característica essencial das operações escrita/leitura numa memoria partilhada é que qualquer operação de leitura que comece depois da conclusão de uma operação de escrita, deve retornar esse valor, ou o resultado de uma outra operação de escrita posterior.

- **Disponibilidade**: um sistema que se mantem disponível durante todo o tempo, significa que, independentemente da existência de falhas na rede, o cliente, eventualmente, receberá uma resposta, ao pedido feito. Esta abordagem não tem restrições no tempo que o serviço demora a terminar, e a resposta pode estar ou não correta.

- **Tolerância à partição**: num sistema distribuído em vários nós, há a possibilidade de existir falhas na comunicação entre os mesmos. Ser tolerante à partição significa que o sistema continuará a trabalhar mesmo sobre falhas na rede, ou seja, o sistema suportará qualquer quantidade de falhas na rede, que não sejam iguais à falha total da mesma.

A diferença entre a disponibilidade e a tolerância à partição está no facto de a disponibilidade significar que o sistema retornará uma resposta, mesmo se um nó falhar,

enquanto a tolerância à partição significa que o sistema continuará a trabalhar, mesmo se existir uma falha na comunicação entre nós (os nós estão a funcionar, mas não comunicam).

Implicações de CAP

É impossível providenciar todas estas características, apenas nos devemos contentar com duas das três, esta é a implicação do teorema de CAP:

- **Consistência / Tolerância à Partição:** imaginando um sistema distribuído por vários nós, se a disponibilidade não é uma condição, alcançar a consistência e a tolerância à partição pode ser feito com a utilização de um nó mestre sincronizado com todos os outros nós do sistema. Para manter integridade da resposta, quando existe alguma operação de escrita, até todos os nós terem processado esta operação, o sistema estará indisponível. Nestes sistemas se acontecer um erro numa parte da rede, o serviço não retornará mais uma resposta, ficando indisponível por um período de tempo. MongoDB é um exemplo destes sistemas.

- **Consistência / Disponibilidade:** uma maneira de obter um sistema consistente e disponível é pôr tudo o que está relacionado com as mesmas transações no mesmo servidor. De facto, se não existe partição, e todos os dados estão guardados no mesmo servidor, o exemplo descrito em cima, segue estes requisitos. Se o único servidor falhar não existirá disponibilidade, mas também não causará inconsistência nos dados.

- **Disponibilidade / Tolerância à Partição:** de forma trivial, um sistema que simplesmente, retorne o mesmo valor, o valor inicial, a todos os pedidos, segue estes requisitos. Se a prioridade é a disponibilidade em detrimento à consistência, pensarmos no exemplo anterior, da secção Consistência/Tolerância à Partição, mas com os nós não sincronizados, não seria necessário esperar que as operações fossem executadas por todos os nós, e o sistema retornaria sempre uma resposta, mesmo que a sua consistência seja infringida. Estes sistemas permitem que as operações de leitura possam ser inconsistentes durante um curto período de tempo, de forma a que o servidor se mantenha sempre disponível.

Disponibilidade vs Consistência

Geralmente num sistema com memória partilhada, os engenheiros não podem abdicar da tolerância à partição. A escolha entre a disponibilidade e a consistência tornou-se um caso particular chave, no estudo dos sistemas distribuídos. Quando se fala destes tipos de sistemas, supostamente, parece haver apenas dois pressupostos plausíveis: sacrificar a disponibilidade ou sacrificar a consistência.

No entanto, Eric Brewer defende que os engenheiros não precisam de escolher entre a disponibilidade e a consistência, existe a flexibilidade para lidar com as partições. Se houver um gerenciamento das partições, é possível otimizar a relação entre a disponibilidade e a consistência, gerando um certo *trade-off* entre todas as três (E. Brewer 2012).

A escolha entre a disponibilidade e a consistência pode ocorrer bastantes vezes dentro do mesmo sistema. O objetivo é maximizar as combinações entre as mesmas que façam sentido para uma aplicação especifica. Não só os subsistemas podem ter diferentes decisões, como a escolha pode alterar consoante a operação, os dados específicos ou o utilizador envolvido (E. Brewer 2012).

Considerando que existem dois nós tolerantes à partição, o nó A e o nó B, e existe uma quebra na comunicação entre ambos. Nesta altura, existe duas opções:

- Cancelar a operação, e dizer que o sistema está em baixo, renunciando a disponibilidade.
- Proceder com a operação, mesmo que os nós não estejam sincronizados, arriscando inconsistência nos dados.

Cabe aos engenheiros decidir qual a opção mais apropriada para o sistema. Muitos sistemas podem não ter um único requisito. Alguns aspetos podem exigir uma forte consistência e outros uma forte disponibilidade. Uma abordagem natural, será dividir o sistema em subsistemas que proporcionam diferentes garantias. Na perspetiva do sistema num todo, pode parecer que os engenheiros renunciaram tanto a disponibilidade como a consistência. Porém, o resultado desta abordagem, frequentemente, origina um sistema que proporciona consistência onde ela é necessária, mas também um sistema que responde aos

pedidos dos utilizadores, mesmo sobre uma rede com más condições. No final, cada parte do serviço proporciona exatamente o que é necessário (Gilbert and Lynch 2014).

2.2 Tipos de Bases de Dados

Nesta fase, já são conhecidas quais as principais vantagens da utilização de sistemas NoSQL, e qual os problemas que estes resolvem. Ao longo dos últimos anos, têm surgido vários tipos de bases de dados não relacionais, que suportam diferentes géneros de dados e que são categorizadas pela maneira como guardam a informação.

Nesta secção vão ser apresentados os diferentes tipos de bases de dados NoSQL existentes, excluindo as bases de dados de Grafos. Este tipo de base dados não vai ser referido, pois tem uma estrutura bastante diferente das restantes, não sendo relevante para o objetivo em questão.

2.2.1 Bases de dados Chave-Valor

Bases de dados Chave-Valor são semelhantes a mapas ou dicionários onde os dados são endereçados por uma única chave. Os valores associados às chaves são isolados e independentes uns dos outros. Devido a esta estrutura de dados bastante simples, bases de dados chave-valor são completamente livres de um esquema. Novos valores, de qualquer tipo, podem ser adicionados durante o tempo de execução sem criar conflitos com quaisquer outros dados guardados e sem influenciar a disponibilidade do sistema. Estes sistemas, são úteis para operações simples, baseadas unicamente em chaves. (Hecht and Jablonski 2011).

Bases de dados Chave-Valor podem ser visualizadas como uma base de dados relacional com múltiplas linhas e apenas duas colunas (Chave e Valor). Ao contrário das chaves, não existe limite no tamanho dos valores, que podem ser tantos objetos, como *hashes*. Maioria das bases dados chave-valor dão prioridade à elevada escalabilidade em vez de consistência (Kaur and Rani 2013). Nenhum destes sistemas suporta um índice ou chave secundária, sendo a modificação e pesquisa da informação feita unicamente a partir da chave primária. (Cattell 2011)

2.2.2 Bases de dados de Famílias de Colunas

Em primeiro lugar, é necessário referir que o termo "família de colunas" é diferente de "bases de dados de colunas". Bases de dados de colunas guardam toda a informação dentro de uma coluna de uma tabela, na mesma localização no disco. Além que, providenciam uma interface SQL para aceder aos dados, MonetDB e Vertica são exemplo destes sistemas (McCreary and Kelly 2014). Um Sistema de família de colunas usa identificadores de linhas e colunas, como chaves para fazer a pesquisa. Quase todas estas bases de dados são inspiradas na BigTable da Google, e são este os sistemas que vamos debater em seguida (McCreary and Kelly 2014).

Segundo (Fowler 2015), estas bases de dados organizam os dados para uma rápida execução de operações de colunas. Possivelmente, a diferença principal para uma base de dados relacional é que cada registro (linha) não necessita de um único valor por coluna. Em vez disso, é possível modelar famílias de colunas. Observando a Figura 2-4, um único registro consiste em um campo ID, uma família de colunas para a informação do *Customer* e outra família de colunas para a informação do *order item*. Uma grande vantagem perante as bases de dados relacional, é que é possível obter toda a informação relacionada, usando um único ID, em vez de utilizar interrogações mais complexas. Sendo possível retornar toda a informação sobre a *order* selecionando uma única linha de colunas.

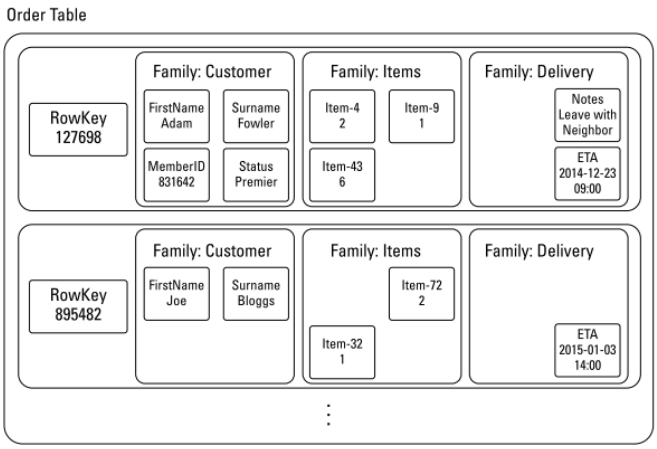

Figura 2-4 Exemplo de um registro numa base de dados de família de colunas, onde toda a informação é armazenada numa única linha de colunas (Fowler 2015)

Colunas podem ser agrupadas em famílias, que são especialmente importantes para organizar e particionar os dados. Colunas e linhas podem ser adicionadas flexivelmente

durante o tempo de execução, mas no caso das famílias de colunas têm que ser predefinidas frequentemente, que leva a uma menor flexibilidade em comparação com as bases de dados chave-valor ou bases dados de documentos (Hecht and Jablonski 2011). As famílias de colunas ou grupo de colunas, são distribuídas pelos múltiplos nós, e são uma simples maneira para o cliente indicar que colunas são melhor serem guardadas juntas. A escalabilidade é feita dividindo as linhas e as colunas por múltiplos nós. As linhas são divididas pelos vários nós, através da partição da chave primária. Normalmente os dados são divididos em um intervalo de valores, em vez de com funções *hash*, isto significa que as *range queries* não precisam aceder a dados em diferentes nós (Cattell 2011).

Devido ao formato das tabelas, as bases de dados orientadas a colunas têm uma representação gráfica semelhante às bases de dados relacionais. Umas das diferenças principais, reside na capacidade de tratar valores nulos. Considerando um caso, com diferentes tipos de atributos, se o *dataset* não tivesse valores associados a alguns atributos, nas bases de dados relacionais seriam armazenados valores nulos em cada coluna. Em contraste, as bases dados orientadas a colunas, se for necessário, apenas guardam um par chave-valor em uma linha. Isto é o que se chama ser esparso e que faz este género de bases de dados bastantes eficiente em domínios com grandes quantidades de dados e diferentes atributos. Este modelo é utilizado para aplicações que lidam com grandes quantidades de dados guardados em grandes *clusters*, porque os dados podem ser particionado eficientemente (Hecht and Jablonski 2011).

2.2.3 Bases de Dados de Documentos

Bases dados de Documentos, como o nome indicam são bases de dados que guardam a informação em estruturas de documentos. Comparado com bases de dados relacionais, os documentos correspondem aos registros (linha), numa tabela. No entanto, um documento é muito mais que uma linha, isto porque, documentos podem conter múltiplos tipos de dados complexos, enquanto que cada registro numa tabela tem o mesmo número de campos. (Kaur and Rani 2013)

Normalmente, os documentos têm o formato JSON, contudo, também existem bases dados documentos que guardam os dados no formato XML, Figura 2-5. Para este âmbito, apenas serão relevantes os documentos com o formato JSON. Esta estrutura é representada

por múltiplos pares chave: valor. Segundo (Hows et al. 2013), "cada chave não passa de um rótulo, é aproximadamente equivalente ao nome que se pode dar a uma coluna em uma RDBMS".

XML

```
<empinfo>
   <employees>
      <employee>
         <name>James Kirk</name>
         <age>40></age>
      </employee>
      <employee>
         <name>Jean-Luc Picard</name>
         <age>45</age>
      </employee>
      <employee>
         <name>Wesley Crusher</name>
         <age>27</age>
      </employee>
   </employees>
</empinfo>
```

JSON

```
{  "empinfo" :
      {
         "employees" :  [
         {
            "name" : "James Kirk",
            "age" : 40,
         },
         {
            "name" : "Jean-Luc Picard",
            "age" : 45,
         },
         {
            "name" : "Wesley Crusher",
            "age" : 27,
         }
                        ]
      }
   }
```

Figura 2-5 Diferença entre o formato XML e o formato JSON, (Hoi 2017)

Os documentos não precisam de seguir a mesma estrutura, existindo a possibilidade de dentro da mesma coleção documentos terem estrutura diferentes (Cattell 2011). Podendo adicionar vários tipos de dados. Isto porque, os documentos podem ser alterados individualmente, independentemente dos outros documentos.

Nestes sistemas é permitido embeber documentos. Documentos embebidos são documentos, com o seu próprio esquema, que fazem parte de outro documento, ou seja, é um documento dentro de um documento. Considerando a Figura 2-6, observa-se que, ao contrário das bases de dados relacionais, as bases de dados Mongo podem guardar a informação de todas as tabelas, por exemplo a *comments* e *tags*, em diferentes documentos embebidos dentro do documento *Blog Post*.

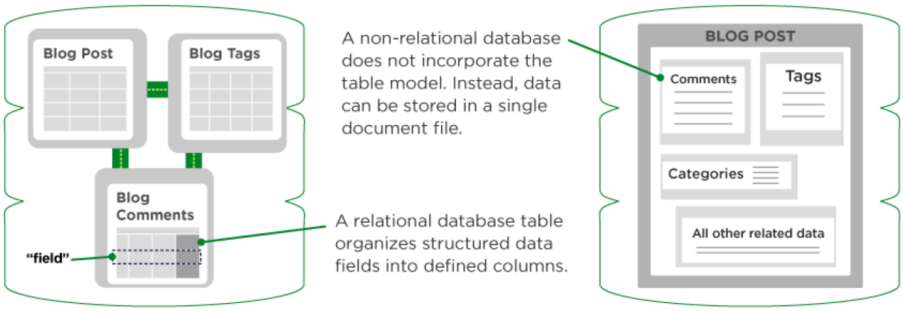

Figura 2-6 Exemplo onde os documentos "Comments", "Tags", "Categories", "All other related data" estão embebidos dentro do documento "Blog Post" (Wodehow n.d.)

2.3 MongoDB

Criado pela 10gen e escrito em C++, MongoDB é uma base de dados de documentos que guarda os registros no formato BSON. BSON é uma representação binária de documentos JSON, e na prática não altera a maneira como os dados são tratados. A grande vantagem de utilizar esta estrutura, é tornar o sistema mais rápido, pois é uma estrutura mais fácil para um computador processar e encontrar os documentos. Tem a característica adicional de, ao contrário de JSON, permitir o armazenamento de dados binários (Hows et al. 2013).

Como já foi referido anteriormente, os documentos são formados por múltiplos pares chave: valor, como ilustrado na Figura 2-7. Cada documento contém uma chave especial chamada "_id", que também é única dentro da sua coleção e assim sendo identifica explicitamente cada documento. Se o utilizador não definir um id, MongoDB criará uma chave única, para o documento em questão (Hecht and Jablonski 2011). Documentos MongoDB não necessitam que seja atribuído um determinado valor aos campos, ao contrário das bases de dados relacionais, onde cada campo necessita de ser definido com um determinado valor, mesmo que este seja *null*. Por exemplo, se um empregado não tiver a carta de condução, basta não introduzir o valor associado a esse campo, em vez de coloca-lo a nulo (Hows et al. 2013).

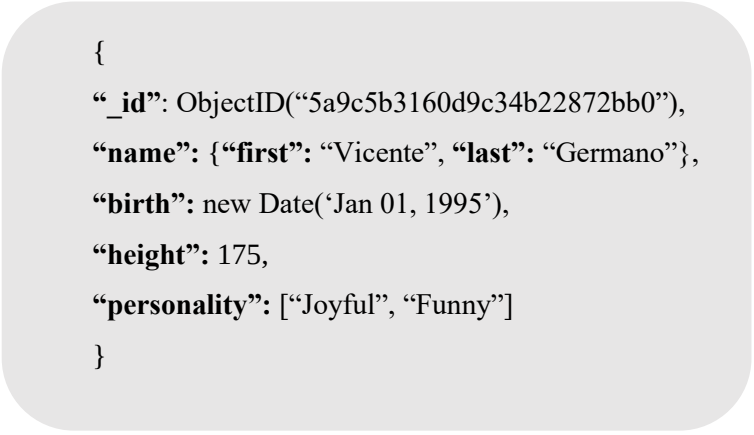

Figura 2-7 Exemplo de um documento MongoDB com formato Chave (Negrito): Valor

No caso de ficheiros que ultrapassem o limite de tamanho de um documento BSON, 16 MB, MongoDB suporta a especificação GridFS, para objetos binários grandes, como, por

exemplo, imagens e vídeos, que em vez de serem guardados num único documento, são divididos em *chunks*, guardando cada *chunk* num documento separado.

Em MongoDB os documentos estão guardados em coleções. Coleções são semelhantes às tabelas do mundo relacional, Figura 2-8, porém bastante mais flexíveis. Ao contrário das tabelas, não são rígidas com a sua estrutura, é possível adicionar quaisquer dados, de diferentes tipos, numa coleção. Se o utilizador não criar uma coleção, esta é criada automaticamente, quando um documento é guardado, esta funcionalidade pode ser relevante quando queremos criar uma aplicação dinâmica (Hows et al. 2013).

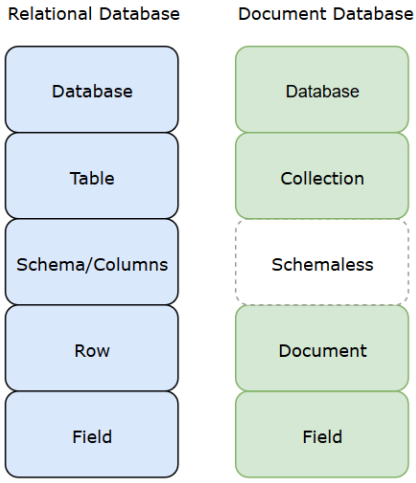

Figura 2-8 Diferença entre a estrutura de uma RDBMS e MongoDB

Nas próximas secções, incluindo o Capitulo três, Linguagem de Interrogação MongoDB, serão apresentadas as principais características desta base de dados. Caso não seja dito o contrário, a informação é baseada nos diferentes capítulos da documentação oficial MongoDB, proveniente do site "MongoDB Manual", (MongoDB 2018a).

2.3.1 Armazenamento dos objetos

Mesmo que as coleções não forcem uma estrutura nos documentos e os dados tenham um esquema flexível, é uma boa prática, que os documentos numa coleção sejam semelhantes, por esta razão, as coleções acabam por ter um esquema implícito. O desafio quando o modelo de dados é delineado, é ter em consideração as necessidades da aplicação, sempre considerando a estrutura inerente dos próprios dados e como a aplicação irá utilizá-los. Existem duas ferramentas que permitem relacionar os dados:

- **As referências,** que guardam as relações entre os dados, como na Figura 2-9. incluindo ligações ou referências de um documento noutro documento. Normalmente, são modelos de dados normalizados.

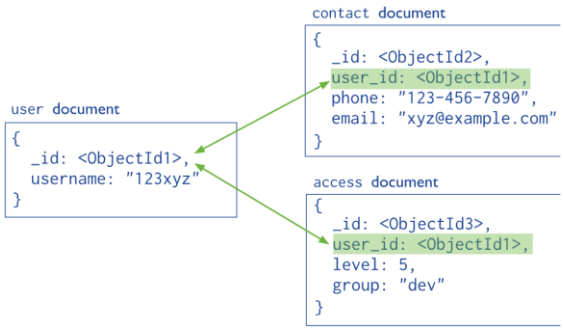

Figura 2-9 Documento a referenciar outro documento (MongoDB 2018e)

- **Dados embebidos** é uma estratégia onde os dados estão relacionados dentro de um único documento, Figura 2-10. É possível aninhar documentos dentro de um campo ou dentro de um *array*. Este modelo de dados é desnormalizado e permite que os dados sejam acedidos numa única operação.

Figura 2-10 Documento embebido (MongoDB 2018e)

2.3.2 Índices

Em contraste com as bases de dados chave-valor, este sistema suporta índices secundários. Os índices são estruturas especiais de dados que armazenam uma pequena porção dos dados de uma coleção, em uma forma fácil de percorrer, suportando a execução eficiente das interrogações. Sem índices, o sistema teria de verificar todos os documentos numa coleção, para selecionar quais os documentos que correspondem à consulta.

Por exemplo, numa loja de jogos, se um cliente pedir um certo jogo, não faz sentido o empregado andar a volta de toda a loja à procura do jogo pretendido. Faz sentido ter um

sistema de indexação, onde o jogo estava associado a um género, o empregado sabendo o género, saberia onde o jogo estava, e facilmente o encontrava. Se um índice apropriado para uma consulta existe, pode ser utilizado para limitar o número de documentos a verificar, como na Figura 2-11. A utilização da condição do score abaixo dos 30, limita o intervalo de documentos que o sistema precisa de procurar.

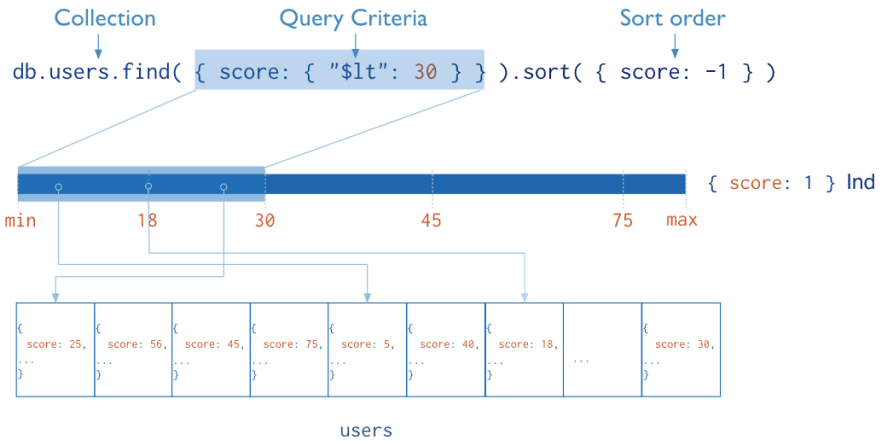

Figura 2-11 Exemplo de um índice secundário, onde o conjunto de resultados é reduzido apenas aos valores onde a pontuação (score) é inferior a 30 (MongoDB 2018g)

2.3.3 Concorrência

A partir da versão 3.2 do MongoDB, a componente, por omissão, responsável pela gestão dos dados, tanto em memória como no disco, e recomendável para novas implementações é a *WiredTiger*. Esta ferramenta faz a gestão da concorrência das operações de escrita ao nível dos documentos. Isto significa que vai ser possível diferentes clientes poderem alterar diferentes documentos de uma coleção ao mesmo tempo.

WiredTiger usa o controlo de concorrência multi-versões, permitindo que múltiplos utilizador vejam diferentes versões dos dados. No inicio da operação, o sistema vai tirar um *snapshot* dos dados, criando um *checkpoint*. Quando é necessário escrever para o disco, *WiredTiger* escreve os dados associados ao *snapshot* de uma maneira consistente ao longo dos vários ficheiros. O *checkpoint* é uma segurança que os dados estão consistentes, anteriormente a esse momento, podendo servidor de pontos de recuperação.

As operações são atómicas ao nível dos documentos, e nenhuma operação de escrita única pode alterar mais que um documento, mesmo que a operação modifique diferentes

documentos embebidos, num único documento. Quando uma única operação de escrita modifica mais do que um único documento numa coleção, ainda assim operam um documento de cada vez.

2.3.4 Replicação

Segundo (Khan and Mane 2013), MongoDB não utiliza a replicação *master/master*, onde dois servidores separados podem receber pedidos de escrita simultâneos. Em vez disso, para suportar grandes quantidades de informação e *streams* de operações, distribui os dados pelos múltiplos servidores, *sharding*. Cada servidor é responsável por atualizar diferentes partes do conjunto de dados. A arquitetura é construída utilizando três componente principais: os nós de fragmentação (*Shard nodes*), Servidores de configuração (*Configuration servers*) e serviços de encaminhamento (*Routing services*) ou mongos, como ilustrado na Figura 2-12 .

1. **Nós de fragmentação**: os nós são responsáveis por guardar os dados. O *cluster* em MongoDB é constituído por um ou mais fragmentos. Cada fragmento consiste em um único nó ou num nó replicado que apenas é constituído pelos dados desse fragmento. Um nó replicado pode consistir em um servidor ou mais servidor, onde um dos servidores atua como o principal, e os outros como secundário (Khan and Mane 2013). A replicação neste sistema é maioritariamente usada para "f*ailover*" (Cattell 2011). No caso de o servidor principal falhar, automaticamente, um servidor secundário torna-se primário. Todas as operações de escrita e de leitura consistente são processadas primeiro pelo servidor principal e todas as operações de leitura "eventualmente consistentes" são distribuídas pelos servidores secundários (Khan and Mane 2013).

2. **Servidores de configuração**: Estes servidores são utilizados para guardar "metadados" e para encaminhar a informação no *cluster*, indicando que dados estão presentes em cada fragmento (Khan and Mane 2013).

3. **Serviços de encaminhamento ou mongos**: São responsáveis por concretizar as tarefas solicitadas pelos clientes. Para processar os diferentes tipos de operações, os mongos enviam pedidos para os nós necessários, e juntam o resultado que posteriormente é enviado de volta para o cliente. Mongos podem correr em paralelo (Padhy, Patra, and Satapathy 2011).

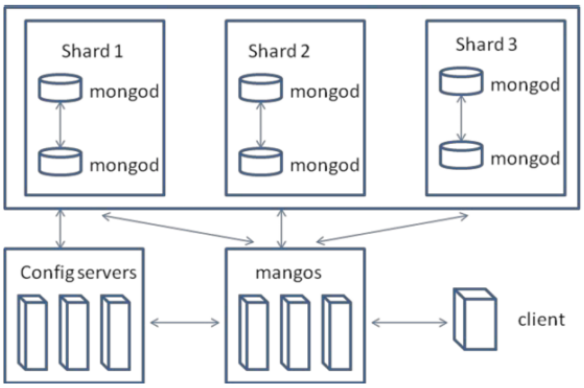

Figura 2-12 Estrutura de replicação MongoDB (Padhy, Patra, and Satapathy 2011)

2.4 Trabalhos Relacionados

Atualmente, já existem sistemas de tradução SQL desenvolvidos, estes tradutores podem ser aplicações web ou desktop, e vão desde os mais simples, onde não é possível converter todas as clausulas SQL, até aos mais complexos, que possibilitam o utilizador interrogar uma base de dados Mongo com qualquer interrogação SQL.

As páginas web (QueryMongo n.d.) e (Klaus.dk 2012), são dois exemplos de tradutores, que permitem aos utilizadores conhecer o comando *find* correspondente ao comando *select* introduzido. Estes dois tradutores, não têm nenhuma ligação com uma estrutura de dados Mongo, o utilizador introduz o comando *select* pretendido, e o sistema apenas retorna o comando *find* equivalente. São dois tradutores simples, não englobando todas as clausulas SQL, como é o caso do *join, group by, having ou in*.

Nos tradutores mais complexos estão incluídos os programas (Studio3T n.d.), (Apache Drill n.d.), (SlamData 2017) ou (NoSQL Booster n.d.). A partir do mapeamento da estrutura MongoDB, é possível ao utilizador interrogar uma base de dados MongoDB com linguagem SQL. A interrogação SQL é traduzida para o comando Mongo equivalente, sendo

possível converter todas as clausulas SQL, incluindo o *join*. Nestes sistemas o utilizador necessita de conhecer a estrutura da base de dados MongoDB previamente.

Apesar de já existirem sistemas de tradução de interrogações SQL para MongoDB, a grande diferença perante o tradutor que vai ser desenvolvido, está no facto das aplicações que têm esta funcionalidade, necessitarem que o utilizador tenha, previamente, algum conhecimento sobre a estrutura da base de dados MongoDB.

O tradutor desenvolvido, tem como público alvo, utilizadores que não conhecem MongoDB, dando-lhes a oportunidade de conseguir aceder aos dados, sem se terem de preocupar como esta base de dados é estruturada. O utilizador não necessita de conhecer a estrutura das coleções ou dos documentos que as compõem, porque a aplicação fará a transposição do modelo Mongo para o modelo relacional, do qual já está habituado.

Capítulo 3 - Linguagem de Interrogação MongoDB

Em MongoDB, não é utilizado a linguagem SQL, em vez disso, por omissão, o equivalente ao *select* é a interrogação *find. N*o entanto, existem muitas outras interrogações incluindo a *aggregate*. Estas duas interrogações são as relevantes para o tema em questão e serão estas estudadas detalhadamente nas seguintes secções.

3.1 Find

A maneira mais simples de obter os dados de documentos dentro das coleções, é a partir da função *find*. Segundo (MongoDB 2018f), esta função, seleciona os documentos numa coleção ou numa *view* e retorna um cursor para os documentos selecionados. A função *find* é constituída por dois parâmetros, Figura 3-1, especificados no formato JSON. O primeiro parâmetro, define a condição de seleção, usada para consultar os dados, ou seja, apenas serão retornados os documentos que cumpram a condição imposta. Enquanto que, o segundo parâmetro, chamado projeção, é utilizado para reduzir o conjunto de resultados num subconjunto. Especialmente nos casos em que existem documentos com tamanhos grandes, o uso deste parâmetro pode ser bastante vantajoso.

```
db.collection.find( { condition }, { projection } )
```

Figura 3-1 Formato de uma interrogação find

Ambos os parâmetros são opcionais, sendo possível interrogar a base de dados, sem nenhum deles. Nesta situação, serão retornados todos os campos de todos os documentos que constituem a coleção. No entanto, tipicamente, não é necessário obter toda a informação de todos os documentos. Em vez disso, o pretendido poderá ser, retornar apenas um determinado tipo de documentos. Para encontrar documentos que cumpram um certo critério, como foi mencionado anteriormente, é necessário especificar a condição pretendida no primeiro parâmetro do find. Supondo que existe uma coleção que guarda a informação sobre músicas, Figura 3-2.

Figura 3-2 Exemplo de uma coleção de músicas

Se o pretendido é, por exemplo, obter as músicas do género Fado, então a interrogação correspondente seria a seguinte, Figura 3-3:

```
db.music.find( { genre : “Fado” } )
```

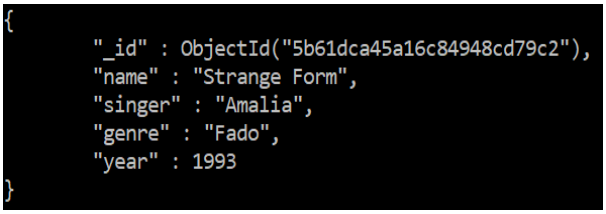

Figura 3-3 Interrogação necessária para obter as músicas do género Fado, juntamente com o resultado da mesma

Porém, se apenas é pretendido conhecer o nome dos cantores de músicas do género Fado, seria necessário especificar o segundo parâmetro da função *find*, com o campo *singer* seguido do valor 1, como ilustrado na Figura 3-4. Nesta situação, o resultado será reduzido apenas aos campos *singer* e id, ignorando o resto dos campos. O campo id é incluído porque, em MongoDB, independentemente da interrogação utilizada, por omissão, é sempre retornado o id dos documentos. Assim sendo, caso não seja pretendido conhecer este campo, e necessário especificar explicitamente “id: 0”.

```
db.music.find( { genre : “Fado” } , { singer : 1} )
```

Figura 3-4 Interrogação necessária para obter as músicas do género Fado, excluindo todos os campos exceto o campo singer, juntamente com o resultado da mesma

Existe ainda a possibilidade de remover qualquer campo, ou campos, do conjunto de resultados. Por exemplo, se o utilizador considerar que o ano em que foram criadas as músicas é irrelevante, tem a possibilidade de excluir esse campo, especificando o campo "*year"*, seguido do valor 0, Figura 3-5. Contudo, é importante mencionar que, segundo (MongoDB 2018f), uma projeção não pode conter tanto uma inclusão como uma exclusão de campos em simultâneo, à exceção do campo id. Nesta situação, o resultado correspondente incluiria todos os campos, exceto o campo *year*.

```
db.music.find( { genre : "Fado" } , { year : 0} )
```

```
{
        "_id" : ObjectId("5b61dca45a16c84948cd79c2"),
        "name" : "Strange Form",
        "singer" : "Amalia",
        "genre" : "Fado"
}
```

Figura 3-5 Interrogação necessária para obter as músicas do género Fado, incluindo todos os campos exceto o campo year, juntamente com o resultado da mesma

3.1.2 Operadores

MongoDB oferece uma enorme quantidade de operadores, que facilitam o utilizador a obter os documentos que deseja. Nas próximas secções, serão estudados alguns dos, mais relevantes, operadores disponíveis em MongoDB. Assim sendo, aconselha-se, vivamente, visitar a página oficial MongoDB, para conhecer tudo que esta base de dados tem para oferecer.

Operadores de comparação

Frequentemente, é pretendido obter apenas os documentos que sejam constituídos por campos com valores dentro de um determinado intervalo. Os operadores *$lt* (antes de), *$lte* (antes de inclusivo), *$gt* (maior que), *$gte* (maior que inclusivo), são operadores de comparação, utilizados para tal. Por exemplo, se o pretendido é conhecer apenas as músicas superiores ao ano 2000, seria necessário especificar o campo ano seguido do operador *$gt* com o valor 2000, Figura 3-6.

```
db.music.find( { year : { $gt: 2000 } } )
```

```
{
        "_id" : ObjectId("5b61dc955a16c84948cd79c0"),
        "name" : "Sound Of Freedom",
        "singer" : "John",
        "genre" : "Piano",
        "year" : 2010
}
{
        "_id" : ObjectId("5b61dc9c5a16c84948cd79c1"),
        "name" : "The Jam",
        "singer" : "Paul",
        "genre" : "Rock",
        "year" : 2007
}
```

Figura 3-6 Interrogação necessária para obter as músicas com ano superior a 2000, juntamente com o resultado da mesma

Para além destes operadores, MongoDB oferece múltiplos operadores de comparação, como é o caso dos operadores $*in* e $*ne*. O operador $in, seleciona os documentos que tenham um valor igual a um dos valores especificados no *array* do operador. Por exemplo, se o desejado for conhecer todos os documentos com o campo género igual a Rock ou Piano, seria necessário especificar o campo *genre* juntamente com o operador $in com os valores “Rock” e “Piano”, Figura 3-7.

```
db.music.find( {genre : { $in: [“Rock”, “Piano” ] } } )
```

```
{
        "_id" : ObjectId("5b61dc955a16c84948cd79c0"),
        "name" : "Sound Of Freedom",
        "singer" : "John",
        "genre" : "Piano",
        "year" : 2010
}
{
        "_id" : ObjectId("5b61dc9c5a16c84948cd79c1"),
        "name" : "The Jam",
        "singer" : "Paul",
        "genre" : "Rock",
        "year" : 2007
}
```

Figura 3-7 Interrogação necessária para obter as músicas com género igual a Rock ou Piano, juntamente com o resultado da mesma

É importante referir que, este operador pode-se confundir com o operador lógico $*or*, descrito na secção seguinte. Isto porque, em situações onde é pretendido verificar os valores do mesmo campo, utilizar o operador $*or* ou o operador $*in* originará o mesmo resultado, porém, segundo (MongoDB 2018c), nestas situações é aconselhável utilizar o operador $*in*. Quando é necessário verificar diferentes campos, então o ideal é utilizar o operador $*or*.

O operador *$ne*, ou *note equal*, é utilizado para selecionar os documentos cujo o valor do campo não é igual ao valor especificado, incluindo também os documentos que não tenham esse campo. Por exemplo, se o utilizador desejar saber todas as músicas que não sejam do género Rock, seria necessário especificar o campo *genre* juntamente com o operador *$ne* com o valor "Rock", Figura 3-8.

```
db.music.find( {genre : { $ne: "Rock" } } )
```

```
{
        "_id" : ObjectId("5b61dc955a16c84948cd79c0"),
        "name" : "Sound Of Freedom",
        "singer" : "John",
        "genre" : "Piano",
        "year" : 2010
}
{
        "_id" : ObjectId("5b61dca45a16c84948cd79c2"),
        "name" : "Strange Form",
        "singer" : "Amalia",
        "genre" : "Fado",
        "year" : 1993
}
```

Figura 3-8 Interrogação necessária para obter as músicas que não sejam do género "Rock", juntamente com o resultado da mesma

Operadores Lógicos

Os operadores lógicos, como o nome indicam, são utilizados para fazer operações lógicas entre as diferentes expressões. O $*and*, $*or* e $*not* estão incluídos nestes operadores. O operador $*and* e o operador $*or*, são especificados por um *array*, constituído pelas múltiplas expressões pretendidas. No caso do $*and* serão retornados os documentos que satisfação todas as expressões especificadas no *array*. Enquanto que, no caso do $*or*, serão retornados os documentos que satisfação pelo menos uma das expressões impostas. Supondo que o pretendido é conhecer todas as músicas superiores ao ano 2000, e do género piano, seria necessário especificar o operador $*and* seguido das expressões "{year : {$gt : 2000}" e {genre: "Piano"}, Figura 3-9.

```
db.music.find( {$and: [{year: {$gt: 2000 } }, {genre: “Piano” } ] } )
```

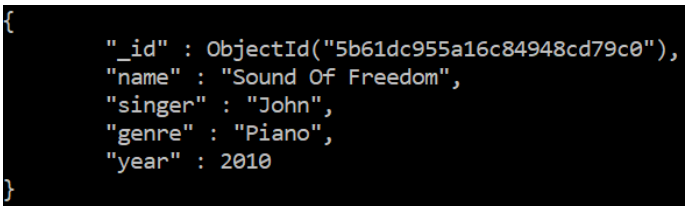

Figura 3-9 Interrogação necessária para obter as músicas que sejam do género Piano e superiores ao ano 2000, com a utilização do operador $and, juntamente com o resultado da mesma

Em certas situações, pode-se gerar confusão se devesse ou não utilizar o operador $*and*, isto porque, em MongoDB, está implícito uma operação *and*, ao especificar as várias expressões separadas, apenas, por virgulas. Considerando a interrogação anterior, mas separando as expressões por virgulas, como representado na Figura 3-10, o resultado originado seria igual.

```
db.music.find( { year : { $gt : 2000 }, genre : “Piano” }  )
```

Figura 3-10 Interrogação necessária para obter as músicas que sejam do género Piano e superiores ao ano 2000, utilizando a operação AND implícita

Assim sendo, segundo (MongoDB 2018b), “usar um AND explícito com o operador $ é necessário quando o mesmo campo ou operador tiver que ser especificado em várias expressões”. Se o pretendido for conhecer, por exemplo, as músicas que sejam do género fado ou piano, “{$or: [{genre: “Fado”}, {genre: “Piano”}]}”, e que sejam do ano superior a 2000 ou que o cantor comece com a letra “J”, “{$or: [{year: {$gt: 2000}}, {name: /^J/}]}”, não seria possível utilizar o *and* implícito porque o operador $*or* é utilizado mais que uma vez. Deve-se utilizar o operador $*and* sempre que existir várias condições associados ao mesmo campo ou operador e não puder combina-las num único objeto.

O operador $*not*, retorna os documentos que não satisfaçam a condição pretendida, incluindo os documentos que não contenham o campo. É importante mencionar que, enquanto o operador $*ne* corresponde à negação de um determinado valor, o operador $*not* não verifica os campos independentemente. Este operador é utilizado para negar o resultado de outro operador. Supondo que é utilizada a interrogação ilustrada na Figura 3-11, seriam retornados todos os documentos que contenham o campo ano inferior a 2000 ou que o campo

ano não exista. Utilizar o operador $*not*, em vez de {$lte : 2000}, está no facto de que, com a utilização do *not*, serão incluídos, também, os documentos que não contenham o campo ano, ao contrário de com a utilização do $*lte*.

```
db.music.find( { year : { $not: { $gt : 2000 } } }  )
```

```
{
        "_id" : ObjectId("5b61dca45a16c84948cd79c2"),
        "name" : "Strange Form",
        "singer" : "Amalia",
        "genre" : "Fado",
        "year" : 1993
}
```

Figura 3-11 Interrogação necessária para obter as músicas que sejam de anos inferiores ou iguais a 2000 utilizando a negação do operador $gt, juntamente com o resultado da mesma

3.1.2 Expressões Regulares

MongoDB permite a utilização de expressões regulares para encontrar padrões em *strings*. Existem duas maneiras para especificar um padrão em Mongo, a utilização do operador $*regex* ou a utilização da sintaxe {campo: /padrão/}, dependendo da situação, poderá ser necessário optar por uma, em vez da outra. Sempre que for utilizado o operador $in, não é possível utilizar o operador $*regex*. Por outro lado, para incluir uma expressão regular, num conjunto de condições separadas por virgulas, *and* implícito, é necessário utilizar o operador $*regex*. Dando um caso, onde o utilizador pretende conhecer todas as músicas dos cantores que comecem com a letra "A", seria necessário especificar a interrogação com a expressão {singer : /^A/} ou a expressão {singer: { $regex: "^A"} }, Figura 3-12. Contudo, se o desejado fosse, por exemplo, conhecer as músicas que começassem com "A", mas não incluísse a cantora Amália, seria necessário utilizar o operador $*regex*, juntamente com o operador $*nin*, porque as condições estariam separadas pelo um *and* implícito, Figura 3-13.

```
db.music.find({ singer: /^A/ } )
```

```
db.music.find({ singer : { $regex: '^A' } } )
```

Figura 3-12 Interrogação necessária para obter as músicas que o cantor comece com a letra "A"

```
db.music.find({ singer : { $regex: '^A', $nin [ "Amalia" ] } } )
```

Figura 3-13 Interrogação necessária para obter as músicas que o cantor comece com a letra "A", não incluído a cantora Amália, utilizando o operador $regex

3.1.3 Documentos Embebidos

Quando se começa a trabalhar com documentos com estruturas mais complexas, como é o caso de documentos embebidos, o processo de interrogação torna-se diferente. Na Figura 3-14, apresenta-se uma coleção de livros que guarda os detalhes sobre os mesmos, incluindo o seu autor.

```
{
        "_id" : ObjectId("5b7ae0d30eb03aa0dd8ea8d3"),
        "IDLivro" : 27,
        "Titulo" : "Uma Ceia De Amor",
        "Editora" : "Dom Quixote",
        "Autoria" : {
                "IDAutor" : "Tournier",
                "Nome" : "Michael Tournier",
                "Nacionalidade" : "Françês"
        }
}
{
        "_id" : ObjectId("5b7ae0d30eb03aa0dd8ea8d4"),
        "IDLivro" : 28,
        "Titulo" : "O Caso Morel",
        "Editora" : "Bertrand",
        "Autoria" : {
                "IDAutor" : "Fonseca",
                "Nome" : "Ruben Fonseca",
                "Nacionalidade" : "Brasileiro"
        }
}
{
        "_id" : ObjectId("5b7ae0d30eb03aa0dd8ea8d5"),
        "IDLivro" : 29,
        "Titulo" : "Ensaio Sobre A Cegueira",
        "Editora" : "Caminho",
        "Autoria" : {
                "IDAutor" : "Saramago",
                "Nome" : "José Saramago",
                "Nacionalidade" : "Português"
        }
}
```

Figura 3-14 Exemplo de uma coleção de livros

Pensando como nos casos anteriores, se o pretendido for conhecer quais os livros pertencentes ao escritor José Saramago, a interrogação necessária utilizaria a sintaxe {nome: "José Saramago"}, no entanto, nesta situação, não seria retornado nenhum documento. Isto porque, quando precisamos de aceder a dados em documentos embebidos, é necessário utilizar uma diferente notação, definida pela sintaxe "Campo.SubCampo". Neste caso, em

vez de utilizar apenas o campo “nome”, seria necessário especificar antes a chave na qual o documento está embebido, “autoria.nome”, como exemplificado na Figura 3-15. É importante mencionar que sempre que for necessário utilizar esta notação, é necessário utilizar aspas, para especificar o campo pretendido.

```
db.books.find( { “autoria.nome” : “José Saramago” } )
```

Figura 3-15 Interrogação necessária para obter os livros de José Saramago, utilizando a sintaxe “campo.subcampo”

3.1.4 Arrays

Uma das grandes diferenças entre mongo e o mundo relacional, é a possibilidade de puder armazenar dados em *arrays*. Contudo, se o pretendido é aceder a um campo que contém este tipo de estrutura, é necessário ter especial atenção a como fazê-lo. Considerando uma coleção que guarda a informação sobre restaurantes, Figura 3-16.

Figura 3-16 Exemplo coleção de restaurantes

Para aceder aos documentos que tenham pelo menos um dos elementos igual ao valor especificado, é utilizada a sintaxe {campo: valor}. Assim sendo, se o pretendido fosse, por exemplo, conhecer todos os restaurantes do género italiano, seria necessário especificar a chave *tags*, juntamente com o valor *italian*, como exemplificado na Figura 3-17. Nesta

situação, serão retornados todos os documentos que contenham no *array tags*, pelo menos um elemento igual a "*Italian*".

```
db.restaurants.find( { "tags" : "Italian" } )
```

Figura 3-17 Interrogação necessária para obter os restaurantes rotulados "Italian"

Se o pretendido é encontrar documentos que coincidam com múltiplas condições, é possível especificar uma condição para apenas um elemento do *array* ou para uma combinação de elementos. Observando a Figura 3-18, utilizar a expressão "{coord: {$gt : 30, $lt: 40 } }", retornará os documentos onde um elemento do *coord*, seja maior que 30 e outro elemento seja menor que 40, ou que um elemento satisfaça ambas as condições, nesta situação, ambos os documentos seriam retornados.

```
db.restaurants.find( { coord : { $gt : 30, $lt: 40 } } )
```

```
{
        "_id" : ObjectId("5b856e1f8a0ff4ba559060d1"),
        "name" : "La Pasta",
        "tags" : [
                "Italian",
                "Japanese"
        ],
        "coord" : [
                40,
                13
        ]
}
{
        "_id" : ObjectId("5b856e1f8a0ff4ba559060d2"),
        "name" : "Tasca do Joel",
        "tags" : [
                "Portuguese",
                "Italian"
        ],
        "coord" : [
                32,
                40
        ]
}
```

Figura 3-18 Interrogação necessária para obter os restaurantes que tenham coordenadas maior que 30 e menor que 40, juntamente com o resultado da mesma

Contudo, se o pretendido é retornar os documentos que tenham pelo menos um elemento que seja maior que 30 e menor que 40, é necessário utilizar o operador $*elemMatch*, como ilustrado na Figura 3-19. Este operador deve ser usado para especificar várias condições, de modo a que pelo menos um elemento do *array* coincida com todas as condições

impostas. Nesta situação, o resultado seria diferente ao anterior, pois existe apenas um restaurante que tem pelo menos uma coordenada maior que 30 e menor que 40.

```
db.restaurants.find( { coord : { $elemMatch: { $gt : 30, $lt: 40  } } }  )
```

Figura 3-19 Interrogação necessária para obter os restaurantes que tenham pelo menos um elemento com as coordenadas maior que 30 e menor que 40, juntamente com o resultado da mesma

3.1.5 Funções

Para além dos vários operadores, MongoDB disponibiliza várias funções que podem ser utilizadas para controlar o conjunto de resultados. Nesta secção vamos estudar algumas dessas funções.

Sort

O *sort* é uma função, como nome indica, utilizada para ordenar o resultado. Se não for especificado a função *sort*(), não é garantido ordem no resultado da interrogação. No exemplo representado na Figura 3-20, serão retornados todos os documentos por ordem ascendente de idade, isto porque, foi especificado na função *sort* a condição { age : 1}. Se o pretendido fosse ordenar descendentemente, seria necessário alterar o valor 1 para -1. Existe ainda a possibilidade de ordenar por múltiplos campos, utilizando a sintaxe {field : order , field2 : order …}

```
db.collection.find( { } ).sort( { age : 1 } )
```

Figura 3-20 Exemplo da utilização da função sort para ordenar, ascendentemente, pela idade

Limit

A função *limit* é utilizada para limitar o número de documentos que a interrogação retornará. A função requer um parâmetro, esse parâmetro é numero máximo de documentos retornados, Figura 3-21. Se o parâmetro for 0, é equivalente a definir sem limite.

```
db.collection.find( { } ).limit( number )
```

Figura 3-21 Exemplo da utilização da função limit()

Count

O *Count* tem como função contar o número de documentos correspondentes à interrogação. Esta função não efetua a consulta à base de dados, em vez disso, conta os resultados que são retornados dessa consulta. Como as restantes funções, têm que juntar a função no final do método *find*, como ilustrado na Figura 3-22.

```
db.collection.find( { } ).count( )
```

Figura 3-22 Exemplo da utilização da função count()

3.2 Framework Aggregate

Imaginando que existe uma situação onde se pretende conhecer qual o distrito de Portugal que produz maior quantidade de cortiça por ano. Em SQL, provavelmente, a solução ideal seria uma interrogação que começasse com *select* campos, tivesse um *group by* e um *having* com diferentes clausulas. Contudo, como já sabemos, MongoDB é uma base dados que não utiliza a linguagem SQL. Então, é nestas situações, que entra em ação a framework *Aggregation*. Segundo (Houlihan 2014), esta framework é um conjunto de ferramentas analíticas, utilizadas para analisar documentos e coleções numa base de dados mongo. As operações de agregação agrupam os valores de diferentes documentos, podendo executar um conjunto de operações nos dados agrupados para retornar o único resultado.

3.2.1 Pipeline

A estrutura *Aggregation* é baseada no conceito pipeline. O conceito pipeline significa a possibilidade de executar uma operação que recebe um input e retorna um output que será utilizado por outra operação como input e assim adiante. A estrutura *Aggregation* utiliza a

mesma metodologia, é uma serie de transformações, executadas em várias fases começando com uma coleção, que é processada, retornando documentos para a seguinte fase e assim sucessivamente, originando, no final, um único resultado que poderá ser um documento, um cursor ou uma coleção, Figura 3-23, (Houlihan 2014).

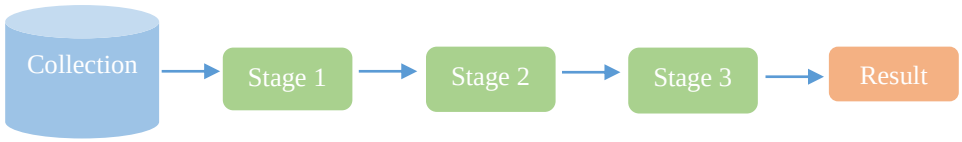

Figura 3-23 Diferentes fases que constituem uma estrutura pipeline

Diferente do *find*, o método *aggregate* é constituído por um único parâmetro, que será o pipeline, Figura 3-24. O pipeline é uma lista formada por múltiplos elementos, cada elemento corresponde a uma fase do processo (*stage*). Cada fase fornece filtros para aceder e transformar documentos que modificam o output. Existem ainda ferramentas para agrupar e classificar documentos por um, ou mais, campos específicos, bem como ferramentas para agregar o conteúdo dos *arrays*. Além disso, nestas fases é possível utilizar operadores para tarefas como calcular a média, ou encontrar o máximo e o mínimo.

```
db.collection.aggregate( [ { stage 1 }, { stage 2 }, … ] )
```

Figura 3-24 Interrogação aggregate constituída pelo pipeline, composto pelas diferentes fases (stages)

3.2.2 Group

Esta fase consiste em agrupar os documentos por uma expressão especificada. O output gerado é formado por um documento por cada agrupamento. Este documento é constituído por um campo id, que distingue cada grupo. O campo id é obrigatório, e pode ser utilizado para agrupar o resultado consoante o, ou os, campos definidos no mesmo. Além disso, é possível definir o id com o valor *null*, para calcular os valores acumulados de todos os documentos como um todo.

Na Figura 3-25, considera-se que existe uma coleção aulas, que guarda a informação sobre a quantidade de alunos presentes em cada uma.

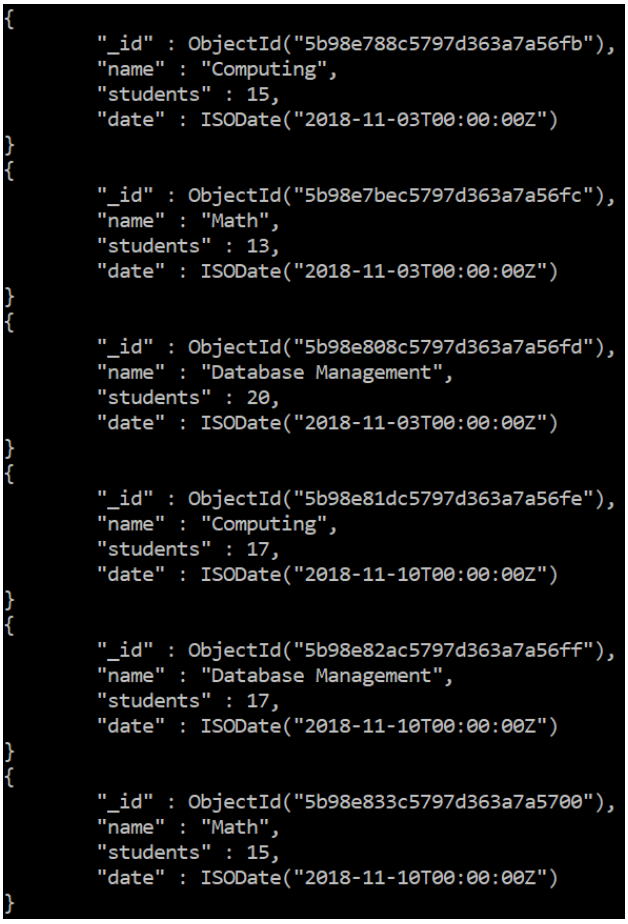

Figura 3-25 Exemplo de coleção de classes de aulas

Supondo que é pretendido conhecer a média de alunos por cadeira, seria necessário agrupar o resultado pelo campo *name*. Assim sendo, o campo id teria que ser definido como $*name*, juntamente com o operador $*avg*, para calcular a media, Figura 3-26.

```
{$group: {_id: “$name”, “avgresult”: {$avg: “$students”} } }
```

```
{ "_id" : "Database Management", "avgResult" : 18.5 }
{ "_id" : "Math", "avgResult" : 14 }
{ "_id" : "Computing", "avgResult" : 16 }
```

Figura 3-26 Fase $group necessária para obter a média de alunos por cadeira

Se o pretendido é conhecer a média geral de alunos de todas as cadeiras, então, seria necessário definir o campo _id com o valor *null*, Figura 3-17. Para além do campo id, os

restantes campos são opcionais, e são calculados usando os operadores de acumulação, como o $*avg*, utilizado no exemplo anterior, Tabela 3-1. É de salientar que na Tabela 3-1, a chave dos operadores estão definidas com "<funcão>Result", porém esta sintaxe apenas é utilizada como exemplo para este caso.

```
{$group: {_id: null, "avgresult": {$avg: "$students"} } }
```

```
{ "_id" : null, "avgResult" : 16.166666666666668 }
```

Figura 3-27 Fase $group necessária para obter a média geral dos alunos

Tabela 3-1 Tabela referente aos diferentes operadores incluídos na fase $group, incluindo descrição e exemplo

Expressão	Descrição	Exemplo agrupado pelo campo "byField"
avg	Retorna a média de valores numéricos	db.col.aggregate([{ $group: {_id: "$byField", avgResult: { $avg: "$field" }}}])
sum	Retorna a soma de valores numéricos	db.col.aggregate([{ $group: {_id: "$byField", sumResult: { $sum: "$field" }}}])
max	Retorna o maior valor da expressão	db.col.aggregate([{ $group: {_id: "$byField", maxResult : {$max: "$field" }}}])
min	Retorna o menor valor da expressão	db.col.aggregate([{ $group: {_id: "$byField", minResult: {$min: "$field" }}}])

3.2.3 Project

A fase "$*project*" é responsável por filtrar os campos desejados que irão passar para as seguintes fases do pipeline. Igual ao parâmetro de projeção do método *find*, o valor 1 é referente à inclusão de campos, enquanto que, o valor 0 é referente à exclusão. Contudo, apesar de terem um comportamento semelhante, não se pode confundir esta fase com o parâmetro referido anteriormente. Para além de selecionar e remover determinados campos, é possível adicionar novos campos ou atribuir valores a campos já existentes. Considerando

uma coleção de corridas que armazena a informação de cada atleta, incluindo idade, a distância percorrida e o tempo decorrido, Figura 3-28.

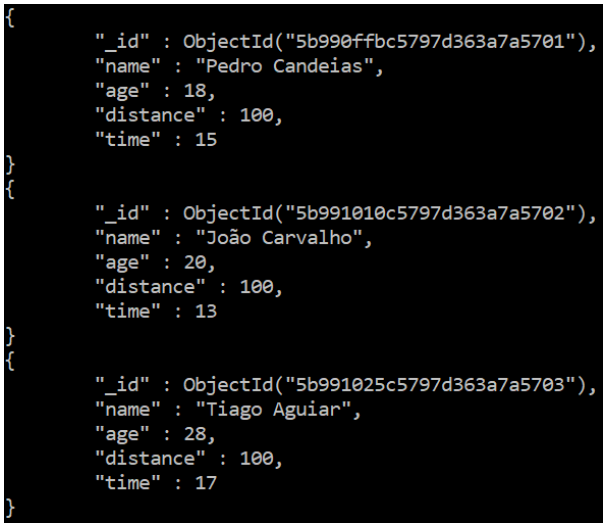

Figura 3-28 Exemplo coleção guarda informação sobre o tempo de corrida de cada atleta

Supondo que é pretendido conhecer a velocidade de cada atleta, ignorando a idade e o id. A utilização do "$*project*" permite criar um novo campo chamado "velocity", Figura 3-29. Este campo será igual à divisão da distância pelo tempo, e utilizará a função mongo "$*divide*", para tal. No final, para além de remover os campos não desejados, foi possível criar um novo campo, a partir de campos já existentes, algo que não seria possível no parâmetro projeção do método *find*.

```
{$project: {_id: 0, name: 1, velocity: {$divide: ["$distance", "$time"]} } }
```

```
{ "name" : "Pedro Candeias", "velocity" : 6.666666666666667 }
{ "name" : "João Carvalho", "velocity" : 7.6923076923076925 }
{ "name" : "Tiago Aguiar", "velocity" : 5.882352941176471 }
```

Figura 3-29 Fase $project utilizada para calcular a velocidade e projetar esse campo juntamente com o nome do atleta

3.2.4 Unwind

A fase "$*unwind*" é utilizada para desmembrar um *array* em cada elemento. Cada documento de saída é o documento de entrada com o valor do campo do *array* substituído pelo elemento. Considerando uma coleção onde são guardadas as múltiplas notas dadas pelos clientes a cada restaurante, a utilização do método *unwind* no *array* grades, {$unwind: "$grades"}, criará um novo documento para cada elemento do mesmo. Observando a Figura

3-30, serão criados dois novos documentos, um para a grade com score 10, e outro para a grade com score 12.

Figura 3-30 Resultado da transformação de um documento em dois novos documentos a partir de cada elemento do array "grades", com a utilização da fase $unwind

Imaginando que é pretendido conhecer a média das notas dadas pelos clientes ao restaurante, o primeiro pensamento seria definir uma interrogação com expressão "{$group: {_id: null, gradesAvg: {$avg: "$grades.score"}}}", porém, nesta situação, o resultado da média seria *null*. Isto porque, quando estamos perante situações como esta, antes de calcular a média, é necessário desmembrar o *array* em múltiplos documentos e só depois agrupar os dados no resultado final, Figura 3-31. A utilização do *unwind* em coleções grandes, pode exceder o limite de memória da estrutura *aggregation*. Assim sendo, nestas situações, é essencial reter o máximo de informação possível com a utilização do "$*project*".

```
{$unwind: “$grades”}, {$group: {_id: null, “gradesAvg”: {$avg: “$grades.score”}}}
```

Figura 3-31 Fases necessárias para calcular a média das notas dadas pelos clientes

3.2.5 Match, Count, Sort e Limit

Para além das fases descritas anteriormente, existem outras fases incluindo o $*Match*, $*Count*, $*Sort* e $*Limit*, Tabela 3-2, que não serão apresentadas tão acentuadamente. Isto porque, estas fases, são bastante parecidas com as funções equivalentes, já explicadas, no método *find*.

Tabela 3-2 Tabela referente às diferentes fases incluídas na framework Aggregate, incluindo descrição e exemplo de cada

Expressão	Descrição	Exemplo
Match	Filtra os documentos, de modo a que passe para a próxima fase, apenas os documentos que cumpram com a condição imposta.	{ $match: { <Query> } }
Count	Retorna um documento que contém a contagem do número de documentos provenientes da fase anterior.	{ $count: “ < $field > ” }
Sort	Classifica os documentos de entrada e retorna os mesmo na ordem classificada.	{ $sort: { < field >: < sort order > } }
Limit	Limita o número de documentos que passaram para a próxima fase do pipeline	{ $limit: < positive integer > }

É importante mencionar que, quando estamos a falar da estrutura *Aggregation*, e por consequente da utilização de um pipeline, relembrando que, nesta estrutura, cada fase recebe o resultado da fase anterior, apesar da disposição das fases ser arbitrária, é essencial ter em consideração a ordem como são introduzidas. Utilizando o exemplo anterior, não faria

sentido agrupar o resultado primeiro e só depois desmembrar o *array* grades, isto porque, a utilização da fase $*group*, primeiramente, originaria um novo documento que não conteria o campo *grades*, esse mesmo documento seria o input da fase $*unwind*, que por sua vez tentaria desmembrar o *array* $*grades* que já não existiria. Outro exemplo relevante é a utilização da fase $*match* o mais cedo possível, pois esta fase limita o número de documentos no pipeline, minimizando a quantidade de dados a processar. Se, por exemplo, o pretendido é conhecer quais os restaurantes de Brooklyn que têm uma média de avaliações maior ou menor que 15. Em primeiro lugar, seriam filtrados todos os documentos que não fossem de Brooklyn, com utilização do *match*, depois seria utilizado a fase $*unwind* para desmembrar o *array grades*, seguidamente, a fase $*group* para agrupar o resultado por nome e calcular a média, e por fim a fase *project*, para fazer uma verificação booleana de quais dos restaurantes têm uma média maior que 15, com a utilização do operador $gt, Figura 3-32.

```
db.restaurants.aggregate([{$match: {borough: "Brooklyn"}}, {$unwind:
"$grades"}, {$group: {_id: "$name", gradesAvg: {$avg: "$grades.score"}}},
{$project: {_id:1, greaterThan15: {$gt: ["$gradesavg",15] }}}])
```

```
{ "_id" : "Orchid Dynasty Restaurant", "greaterThan15" : false }
{ "_id" : "Indian Spice", "greaterThan15" : false }
{ "_id" : "Chelsea'S Juice Factory", "greaterThan15" : true }
{ "_id" : "Uncle Wang Chinese Food", "greaterThan15" : false }
{ "_id" : "Sofia Pizzeria", "greaterThan15" : true }
{ "_id" : "Ab Halal Restaurant", "greaterThan15" : true }
{ "_id" : "Asian City", "greaterThan15" : true }
{ "_id" : "Lefeu Lounge5", "greaterThan15" : false }
{ "_id" : "Aura Bar & Lounge", "greaterThan15" : false }
{ "_id" : "Express Fulton Fried Chicken & Pizza", "greaterThan15" : false }
{ "_id" : "Duo Ro Restaurant", "greaterThan15" : true }
{ "_id" : "Pair Wine And Cheese", "greaterThan15" : false }
{ "_id" : "New Aarpan", "greaterThan15" : true }
{ "_id" : "Combite Creole", "greaterThan15" : false }
{ "_id" : "Denny'S", "greaterThan15" : false }
{ "_id" : "Livingston Manor", "greaterThan15" : false }
{ "_id" : "Chez Alex", "greaterThan15" : true }
{ "_id" : "Mr King Food Kitchen", "greaterThan15" : false }
{ "_id" : "Cocoa Grinder", "greaterThan15" : true }
{ "_id" : "Xi'An Famous Foods", "greaterThan15" : false }
```

Figura 3-32 Interrogação necessária para obter os restaurantes de Brooklyn que têm uma media de avaliações maior ou igual a 15

Capítulo 4 - Regras de Transposição

Para gerar o modelo relacional, o sistema dividirá o processo em duas, principais, fases:

1. Definição das relações para cada campo dos documentos
2. Criação de tabelas, consoante as relações definidas

A primeira fase, consiste em atribuir a cada campo uma relação. Cada relação é definida por um estado, **Um-Um**, **Um-Muitos, Muitos-Um** ou **Muitos-Muitos,** e cada campo poderá assumir apenas um destes estados. O estado será definido consoante a relação do campo, em questão, com os outros campos do mesmo documento, ou com o mesmo campo em documentos diferentes. Apenas serão relevantes os campos que sejam documentos embebidos ou *arrays* de valores, sendo que, os campos singulares serão os campos que compõem cada tabela gerada.

Em primeiro lugar, o sistema percorre todos os documentos da coleção e para cada campo atribuirá um estado. Os estados serão definidos de acordo com o número de ocorrências do campo. Uma ocorrência ocorre quando existir dois ou mais campos iguais dentro do mesmo documento ou coleção. Para ser considerado igual, todos os campos de ambos os documentos embebidos têm de ser iguais. É importante realçar que, se o documento tiver mais um campo que o outro, mesmo que todos os restantes campos sejam iguais, este não será considerado como uma ocorrência. Alem disso, os campos estão sempre associados à sua chave e a chave do documento pai. O documento pai é o documento onde o campo, em questão, está embebido. Isto significa que, apenas vão ser comparados campos (valores), que estejam associadas à mesma chave, e pertençam aos mesmo género pai. Observando a Figura 4-1, apesar do documento embebido, *department*, ser igual em ambos os documentos, como os documentos pais são diferentes, num caso é *militar division* e no outro *company*, não pode ser considerado como uma ocorrência para aquele campo.

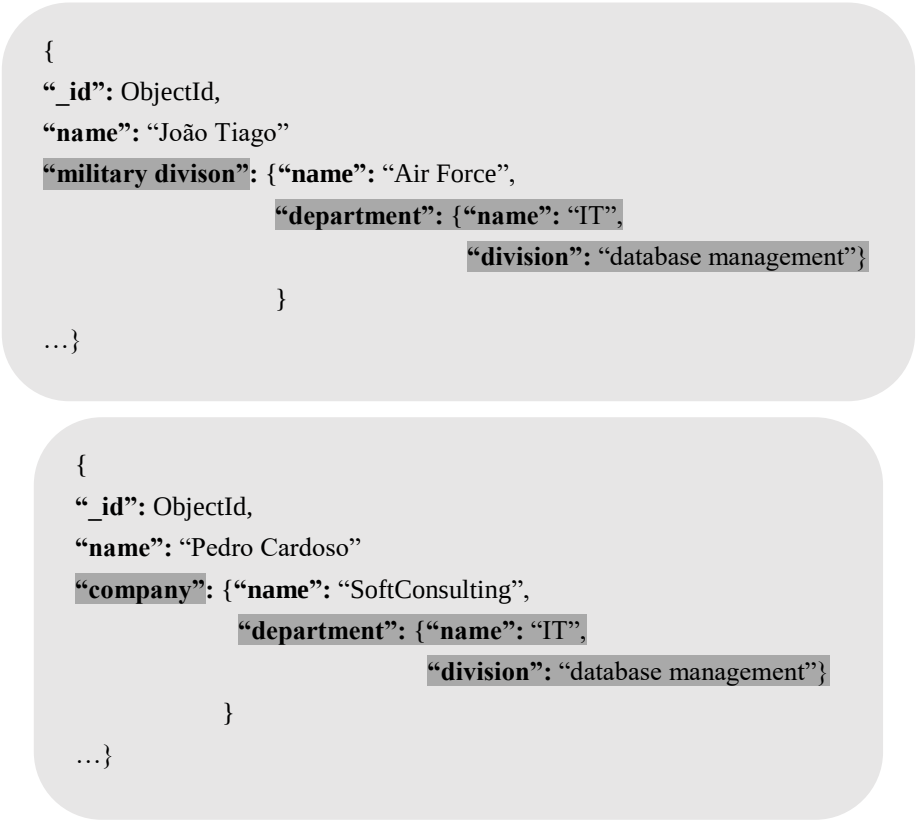

Figura 4-1Mesmo documento, "department", embebido em diferentes documentos, associado a diferentes pais, "militar divison" e "company

As tabelas são ser constituídas pelas chaves dos documentos. Cada valor na tabela corresponde ao valor da chave no documento, ou seja, um documento é como um tuplo numa tabela, e as chaves são semelhantes aos campos (colunas) da mesma. As tabelas criadas terão o nome da chave do campo do qual foram geradas, juntamente com o nome da coleção, em questão. A denominação dos campos de cada tabela consistirá no nome da chave do campo atual, mais o nome de todas as chaves referentes aos níveis anteriores do documento.

Para além dos campos que compunham o documento, é adicionado à tabela um campo identificador. O campo identificador, vai depender do género de documento que gerou a tabela. No caso de ser o documento principal da coleção, então, será acrescentado um campo denominado "id", referente ao campo "_id" do documento. No caso de serem documentos embebidos, como não têm identificador, será criado um campo chamado "link". Este campo serve para ajudar o utilizador a relacionar diferentes tabelas nas interrogações. Não é possível aceder a este campo, porque é um campo somente associativo e não tem dados

associados. É criado, apenas, como chave primária da tabela, que poderá, posteriormente, ser utilizada como chave estrangeira noutra tabela do modelo.

Observando a Figura 4-2, ignorando, para já, qualquer relação com os estados dos campos, o documento principal originaria uma tabela com o nome da coleção, *users*, e cada campo seria nominado com a sintaxe "*users*_<nome do campo>", para além destes campos, seria adicionado o campo id, associado ao campo "_id" do documento *users*. O documento embebido *address*, se fosse necessário gerar uma nova tabela, teria o nome "users_address", os campos teriam a sintaxe "*users_address*_<nome do campo>", e seria adicionado o campo link, para, posteriormente, ser utilizado, se necessário, como chave estrangeira em outras tabelas, Tabela 4-1.

USERS

{**"_id":** objectId,
"name": "Tiago Antunes",
"age": 30,
"address": {**"street":** "Xpto Avenue",
"zipcode": 1234}
}

Figura 4-2 Exemplo de um documento sobre utilizadores

Tabela 4-1 Tabelas geradas pela transposição do documento da Figura 4-2

users
id
users_name
users_age

users_address
link
users_address_street
users_address_zipcode

Após definir o estado, numa segunda fase, o sistema vai voltar a processar todas coleções e consoante o estado atribuído a cada campo, desencadeara diferentes comportamentos. Todos estes comportamentos vão ser descritos nas próximas secções.

4.1 Um-Um

O estado Um-Um, refere-se à situação onde um documento embebido é único ao longo da coleção. Isto indica que, para o mesmo pai, o documento embebido nunca se repete. Numa primeira etapa, quando o sistema estiver a processar o primeiro documento de uma coleção, se existir algum documento embebido, ser-lhe-á atribuído este estado, pois ainda não houve oportunidade de contabilizar mais ocorrências. Considerando uma coleção de três documentos, como ilustrado na Figura 4-3, é possível observar que cada documento pai (editora) tem um único livro, e que esse livro nunca se repete nas outras editoras. Isto significa que, cada editora publicou apenas um livro, e que cada livro foi publicado por uma diferente editora. Assim sendo o estado ideal será One-One.

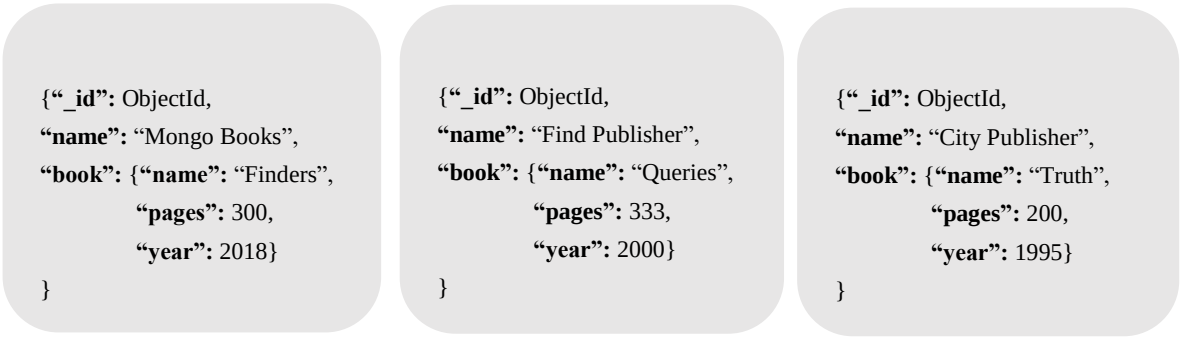

Figura 4-3 Exemplo de uma coleção de editoras constituída por três documentos, onde cada editora publicou um livro diferente

Quando estamos perante esta situação, todos os campos do documento embebido serão acrescentados à tabela relativa ao documento pai. Considerando o exemplo anterior, os campos associados ao documento *book*, seriam acrescentados à tabela editora, como ilustrado na Tabela 4-2.

Tabela 4-2 Tabelas geradas pela transposição do documento da Figura 4-3,

Publishers
id
publishers_name
publishers_book_name
publishers_book_pages
publishers_book_year

4.2 Muitos-Um

Este é o caso onde o mesmo documento embebido é contabilizado mais que uma vez ao longo da coleção. Observando a Figura 4-4, notamos que, o livro *Finders* é publicado tanto pela editora *Mongo Books*, como pela editora *City Publisher*. Assim sendo, como existe um documento (*book*), associado ao mesmo pai (*publishers*), que surge em diferentes documentos, significa que o mesmo livro pode ser publicado por diferentes editoras, e por isso o estado ideal é Muitos-Um.

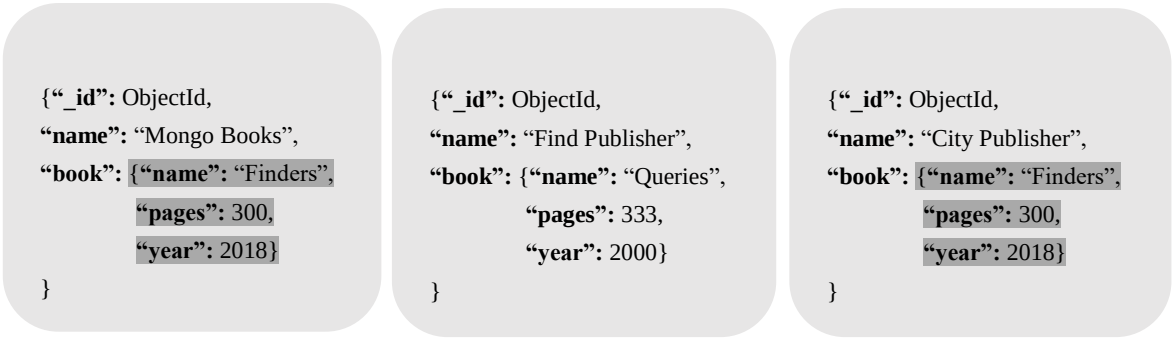

Figura 4-4 Exemplo de uma coleção de editoras constituída por três documentos, onde a diferentes editoras publicaram o mesmo livro

Nesta situação, é criada uma nova tabela referente ao campo definido com estado Muitos-Um, e é acrescentada à tabela associada ao documento pai o campo link da tabela gerada, como chave estrangeira. Seguindo o exemplo anterior, são criadas duas tabelas, Tabela 4-3, uma tabela com o nome da coleção, "publishers", juntamente com a chave *book*, *"publisher_book",* e a tabela *publishers*. A tabela "publisher_book" constituída pelo identificador link e por todos os campos que constituem o documento *book.* Enquanto que, a tabela publishers, é composta pelos campos "id", "*publisher_name*" e a chave estrangeira "*publisher_book_link*", referente à chave primária, link, da tabela "*publishers_book*".

Tabela 4-3 Tabelas geradas pela transposição do documento da Figura 4-4

publishers
id
publishers_name
publisher_book_link

publishers_book
link
publishers_book_name
publisher_book_pages
publisher_book_year

4.3 Um-Muitos

Sempre que alguma chave está associada a um campo do tipo *array*, o estado, inicialmente, atribuído é Um-Muitos. Salientar que, ao contrário das situações anteriores, onde apenas eram relevantes os campos que fossem documentos embebidos, neste caso, quando estamos a falar de *arrays*, independentemente, do mesmo ser constituído por documentos embebidos, ou por qualquer outro valor, é criada uma nova tabela. Isto porque, em ambas as situações, a relação entre o documento pai e o campo é de um para muitos. Considerando a coleção da Figura 4-5, observa-se que tanto a editora *Mongo Books*, como a editora *Find Publisher*, publicaram dois livros cada uma, e que a editora *City Publisher* está rotulada como uma editora do género escolar (*Educational*) e do género científico (*Science*). Pode-se concluir, então, que uma editora pode publicar mais do que um livro e ser rotulada por mais do que o género, assim sendo, o estado atribuído a ambas as chaves, *books* e *genres*, é Um-Muitos. É importante relembrar que, como ilustrado na Figura 4-5, apesar de nem todas as editoras terem publicado mais do que um livro ou terem sido rotuladas por um, ou mais géneros, basta existir uma ocorrência num único documento que defina o estado Um-Muitos, então, este será o estado ideal para o campo em questão.

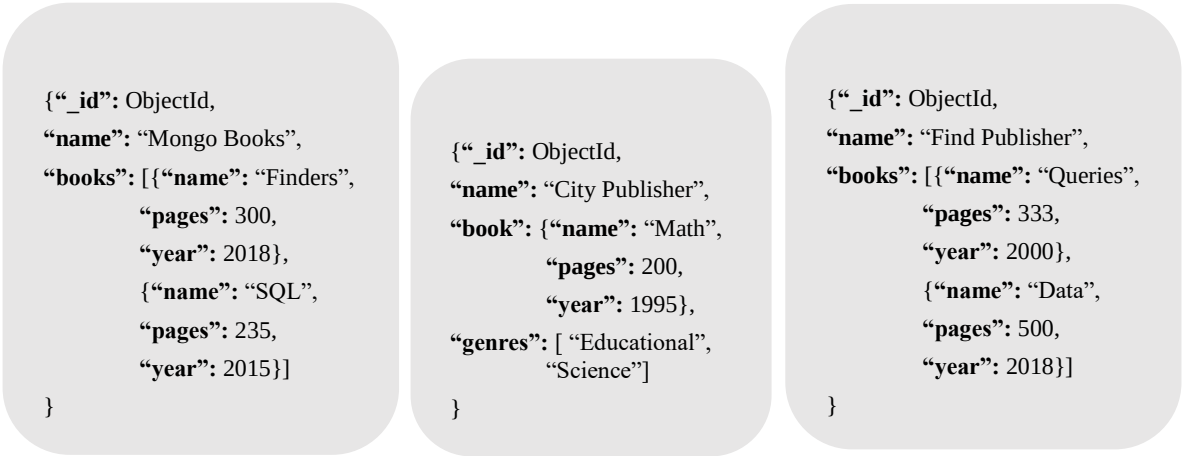

Figura 4-5 Exemplo de uma coleção de editoras constituída por três documentos, onde a cada editora pode publicar múltiplos livros

No caso dos valores do *array* não serem documentos, é criada uma tabela formada por um único campo que irá representar os valores guardados nesse *array*. No caso de serem documentos embebidos, a tabela é formada pelos campos que constituem o documento. Em

ambos os casos, além dos restantes campos, é adicionado um campo referente à chave primária da tabela relativa ao documento pai, como chave estrangeira.

Continuando com exemplo anterior, Figura 4-5, são geradas três tabelas, a tabela *publishers*, composta pelos campos id e *publishers_name*, a tabela *publishers_books*, composta pelos campos link, *publishers_books_name*, *publishers_books_pages*, *publishers_books_year* e a tabela *publishers_genres*, composta pelo campo link e *publishers_genres*. Tanto a tabela *publishers_genres* como *publishers_books*, teriam, como chave estrangeira, a chave primária da tabela "publisher", campo id, Tabela 4-4.

Tabela 4-4 Tabelas geradas pela transposição do documento da Figura 4-5

publishers
id
publishers_name

publishers_books
link
publishers_books_name
publishers_books_pages
publishers_books_year
publishers_id

publishers_genres
link
publishers_genres
publishers_id

4.4 Muitos-Muitos

O estado Muitos-Muitos, representa a situação em que um documento contem um *array* de valores e esses valores surgem em diferentes documentos, ao longo da coleção. Considerando a Figura 4-6, observa-se que a editora Mongo *Books* publicou múltiplos livros, incluindo o livro *Finders*. Em primeiro lugar, poder-se-ia pensar que o estado ideal seria o Um-Muitos, contudo, o mesmo livro, *Finders*, foi publicado, também, pela editora *Find Publisher*. O mesmo se sucede com o *array* dos géneros, onde o género *Educational*, surge tanto na editora *City Publisher* como na editora *Find Publisher*. Assim sendo, nesta situação, o estado ideal é Muitos-Muitos, isto porque, uma editora pode publicar vários livros e ser rotulada por vários géneros, porém, o mesmo livro ou mesmo género, pode ser publicado ou rotulado, também, por diferentes editoras.

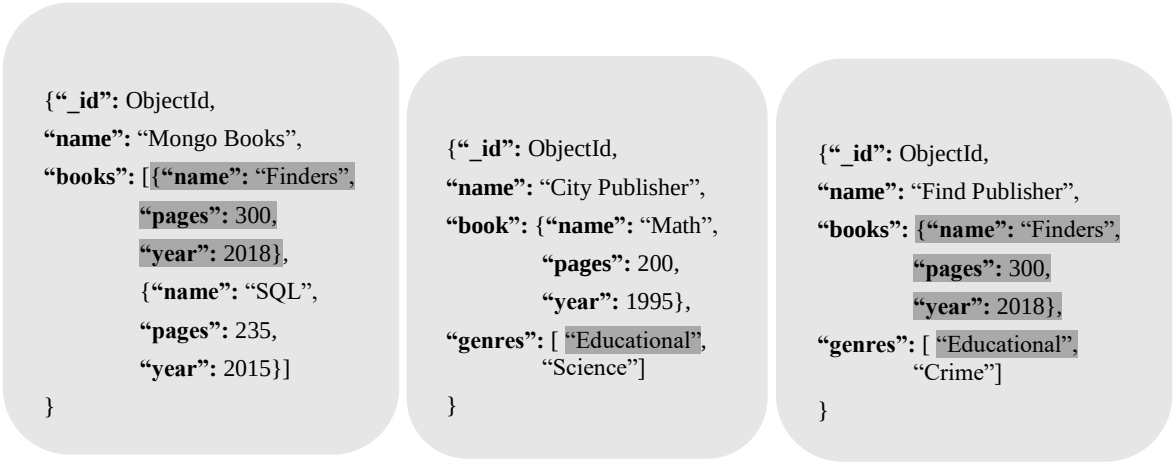

Figura 4-6 Exemplo de uma coleção de editoras constituída por três documentos, onde a cada editora pode publicar múltiplos livros e o mesmo livro pode ser publicado por diferentes editoras

Quando se está perante esta situação, é criada uma tabela associativa, com o nome da tabela pai, juntamente com os caracteres "__" (dois *underscores*), mais o nome da tabela referente ao campo do *array*. A tabela associativa é composta pelo campo *link*, mais os campos referentes às chaves primárias da tabela pai e da tabela associada ao campo *array*.

Considerando o exemplo anterior, Figura 4-6, são criadas, para além das tabelas *publishers_genres* e *publishers_books*, Tabela 4-5, duas tabelas associativas, ilustradas na Tabela 4-6. A tabela "*publishers__publishers_books*" representa a relação entre a editora e os livros, e é constituída pelo campo link e pelas chaves primárias das tabelas *publishers* e *publishers_books*, definidas com o nome *publishers*_id e *publishers_books_link*, respetivamente. A tabela "*publishers__publishers_genre*", representa a relação entre a editora e os géneros, e é constituída pelo campo link mais os campos referentes as chaves primárias da tabela *publishers* e da tabela *publishers_genres*, designados *publishers_id* e *publishers_genres_link*, respetivamente.

Tabela 4-5 Tabelas geradas pela transposição do documento da Figura 4-6

publishers_books
link
publishers_books_name
publishers_books_pages
publishers_books_year

publishers
id
publishers_name

publishers_genres
link
publishers_genres

Tabela 4-6 Tabelas associativas geradas para relacionar as tabelas da Tabela 4-5

publishers__publishers_books
link
publishers_id
publishers_books_link

publishers__publishers_genres
link
publishers_id
publishers_genres_link

Capítulo 5 - Tradutor de Interrogações

Após a transposição da coleção pretendida, no modelo relacional equivalente, o utilizador, consoante esse mesmo modelo, poderá interrogar a base de dados, a partir de um comando *select*. Este comando é traduzido, pelo sistema, para a interrogação MongoDB correspondente, utilizada, posteriormente, para consultar os dados pretendidos. É relevante mencionar que o sistema foi desenvolvido para processar uma interrogação *select* bem estruturada, ou seja, com as clausulas bem escritas e introduzidas pela ordem correta.

Para traduzir o comando *select,* o sistema divide a interrogação em quatro principais fases, Figura 5-1:

1. Converter a tabela ou tabelas pretendidas, na coleção correspondente.
2. Obter os campos (colunas) pretendidas.
3. Processar a condição associada à clausula *where*.
4. No caso de existir restantes cláusulas, processar as mesmas.

SELECT **FASE 2** FROM **FASE 1** WHERE **FASE 3/4**

Figura 5-1 Processo do Tradutor dividido pelas diferentes fases

Um *select* pode originar dois tipos de interrogações MongoDB, uma interrogação *find* ou uma interrogação *aggregate*. A escolha entre ambas depende de como o *select* é constituído. Por omissão, é utilizado o método *find,* no entanto, em certas situações é necessário utilizar a framework *aggregate*, para obter os resultados pretendidos.

Se o utilizador desejar agregar o conjunto de resultados, com a utilização de funções de agregação, como por exemplo, o *having* ou *group by*, a interrogação Mongo equivalente será uma interrogação *aggregate*. Isto porque, como o nome indica, este tipo de interrogação é obrigatória em situações onde é pretendido agregar os dados num único resultado. Contudo, para além da utilização de funções de agregação, sempre que for necessário aceder a dados que estejam armazenados em *arrays*, também, é necessário utilizar a framework *aggregate*. Isto deve-se ao facto de, nesta estrutura, existir a fase *$unwind*. Considerando uma coleção de restaurantes, onde são guardadas as várias notas dadas pelos clientes, Figura 5-2, o modelo

relacional correspondente terá, para além das restantes tabelas, uma tabela relativa ao campo notas chamada "*restaurants_grades*", Tabela 5-1.

Figura 5-2 Exemplo de uma coleção de restaurantes

Tabela 5-1 Uma das tabelas geradas pela transposição do documento da Figura 5-2

restaurants_grades
link
restaurants_grades_name
restaurants_grades_score

Supondo que o utilizador deseja saber todas as notas que sejam maiores que 10. Com a utilização do método *find*, a base de dados retornará, não só, os dados agrupados por documento, que não é desejado, como retornará todas as notas de um restaurante que tenha pelo menos uma das notas maior que 10, mas que possa ter todas outras menores, Figura 5-3.

```
db.restaurants.find({ “grades.score”: { $gt : 10 } }, {“grades” : 1, “_id” : 0} )
```

Figura 5-3 Interrogação find utilizada para obter os restaurantes que tenham uma avaliação maior que 10

Com a utilização da framework *aggregate*, e por consequente da fase *unwind*, não só os dados são retornados separadamente, como apenas sao retornadas as notas a cima de 10, Figura 5-4.

```
aggregate([{$unwind: “$grades”}, {$match: {“grades.score”: {$gt : 10} } },
{$project: {_id: 0, grades: 1} }])
```

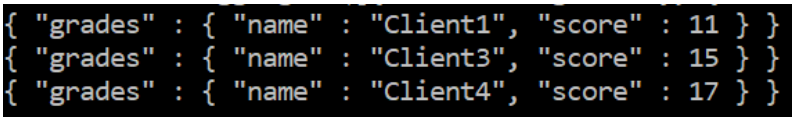

Figura 5-4 Interrogação aggregate utilizada para obter os restaurantes que tenham uma avaliação maior que 10

5.1 Primeira Fase

A primeira fase é responsável por obter a coleção correspondente às tabelas pretendidas pelo utilizador. Como a transposição para o modelo relacional é feita por coleção, todas as tabelas que constituem o modelo estão associadas a uma única coleção. Deste modo, mesmo que o utilizador relacione diferentes tabelas, o comando Mongo equivalente estará associado, sempre, a apenas uma coleção, como ilustrado na Tabela 5-2. Considerando o exemplo da coleção anterior, Figura 5-2, independentemente, se o utilizador desejar conhecer qual o nome dos restaurantes, ou apenas as notas de cada um deles, ou ambas, em todos os

casos, a interrogação Mongo equivalente será especificada com apenas uma coleção, a coleção restaurantes.

Tabela 5-2 Independentemente do número de tabelas utilizadas, estarão sempre associadas a uma única coleção

SQL	FIND
select field from table 1, …, table N	db.collection.find(…)

É essencial mencionar que para interrogações que relacionam diferentes tabelas, o tradutor foi desenvolvido para processar o formato SQL: "select <fields> from table1, table2 where table1.<Chave Primária> = table2.<Chave Estrangeira>", em vez da utilização da clausula *join*.

Para além de obter a coleção correspondente, nesta fase, o sistema guarda, como referência para as fases seguintes, todas as tabelas relativas a dados guardados em *arrays*, de modo a que, nas próximas fases, o sistema saiba que a interrogação correspondente será *aggregate*. Isto devesse ao facto de, como foi demonstrado anteriormente, quando se está perante este tipo de estruturas (*arrays*) o ideal é utilizar a fase *unwind* pertencente à framework *aggregate*.

5.2 Segunda Fase

Esta fase é responsável por apresentar apenas os campos desejados. Primeiramente, é verificado se algum dos campos pretendidos é um campo designado *link*, se for, surgirá uma mensagem de erro, alertando o utilizador que não pode aceder a dados provenientes deste tipo de campos. Isto porque, como foi explicado anteriormente, estes campos não têm dados associados, apenas foram criados para ajudar o utilizador a relacionar as diferentes tabelas do modelo relacional.

Após certificar que nenhum dos campos é um *link*, o sistema verifica se é pretendido que os dados sejam agrupados, com uso de funções de agregação. Se sim, então, o comando Mongo correspondente será uma interrogação de agregação, se não, será utilizado o *find*. Em ambos os casos, é essencial verificar se algum dos campos pretendidos está associado a uma das tabelas referenciadas, na fase anterior. Se estiver, independentemente da utilização de funções de agregação, a interrogação Mongo equivalente utilizará a framework *aggregate*.

No caso de existirem funções de agregação, é utilizada fase *$group*, definida com o campo _id a *null*. Isto porque, para o tradutor, este campo é relevante apenas quando utilizado para agrupar o conjunto de resultados por uma ou mais colunas especificadas na cláusula *group by*, que, caso exista, será processada posteriormente. Para cada função de agregação SQL, considerada, existe um operador Mongo equivalente. Consoante as funções de agregação que o utilizador deseja, estas serão adicionadas ao parâmetro da fase *$group*, com o nome do campo mais *underscore*, juntamente com o nome da função, ficando com a sintaxe "campo_função", como ilustrado na Tabela 5-3.

Tabela 5-3 Expressão $group equivalente a clausula SQL

Clausula SQL	**Expressão $group**
sum(< field >)	{$group: {"_id": null, "field_sum": {$sum: "$< field >"}}}
avg(< field >)	{$group: {"_id": null, "field_avg": {$avg: "$< field >"}}}
max(< field >)	{$group: {"_id": null, "field_max": {$max: "$< field >"}}}
min(< field >)	{$group: {"_id": null, "field_min": {$min: "$< field >"}}}
max(< field >), min(< field >)	{$group: {"_id": null, "field_max": {$max: "$< field >"}, "field_avg": {$avg: "$< field >"}}}}

É importante referir que, quando se trata de uma função *count*, o procedimento é diferente das restantes. Em MongoDB, o método *count* não recebe um campo como argumento, e por isso não existe uma maneira direta de contar o número de ocorrências de um campo em específico. Assim sendo, o tradutor utiliza a condição {*$ne*: *null*}, na fase *$match*, juntamente com a função *$sum*, na fase *$group*, definida com o nome que o utilizador desejar, Tabela 5-4. Esta condição garante que o campo pretendido existe e tem um valor diferente de *null*, enquanto a função *$sum* contabiliza o número de ocorrências. Por

este motivo, no tradutor, apenas é possível fazer um *count* de cada vez, se relacionar dois *counts* em simultâneo, surgirá uma mensagem de erro, alertando o utilizador do sucedido.

Tabela 5-4 Expressão $group equivalente a clausula SQL count

Clausula SQL	Expressão $group
count(<field>)	{$match: {"<field>": {$ne: null }}}, {$group: {"_id": null, "fieldCount": { $sum: 1 }}}

No caso de não ser necessário utilizar a framework *aggregate* e por consequente a utilização do método *find*, nesta situação, os campos pretendidos serão introduzidos no segundo argumento do *find*, como ilustrado na Tabela 5-5. Relembrar que, também nesta situação, é importante ter atenção ao campo id. No caso de o utilizador desejar que o campo id seja apresentado, não é necessário especificar a condição id:1, pois já está implícita.

Tabela 5-5 Expressão find equivalente à expressão SQL

Expressão SQL	Expressão find
select id, field, field2 from tabela	db.collection.find({}, {field : 1, field2 : 1})

5.3 Terceira Fase

Na terceira fase, o sistema verifica a existência, na interrogação *select*, da cláusula *where*, se esta existir, segue-se para o processamento da condição correspondente. Contudo, antes de processá-la, é importante que o sistema saiba qual o tipo interrogação Mongo gerada nas fases anteriores. Se for uma interrogação *aggregate*, a condição é introduzida na fase *$match* e adicionada ao *pipeline*, enquanto que, se for uma interrogação *find*, a condição é adicionada, diretamente, no primeiro argumento do método, como observado na Tabela 5-6.

Tabela 5-6 Condição find equivalente à condição Aggregate

Expressão Find	Expressão Aggregate
db.collection.find({condition})	db.collection.aggregate([{$match:{condition}}])

Existe uma transposição direta de quase todos os operadores lógicos SQL, que podem compor uma condição da clausula *where*, para operadores MongoDB equivalentes, Tabela 5-7. Desde os operadores mais básicos como o "=", ">" ou "<>", até aos operadores mais complexos, como *in*, *like* ou *not*. O operador *between*, ao contrário de todos os outros, é o único que não tem uma transposição direta, sendo necessário utilizar a função *$gte* juntamente com a função *$lte*.

Tabela 5-7 Operadores $group equivalentes aos operadores SQL

Operador SQL	**Operador $group**
field = x	{field: x}
field > x	{field: {$gt: x } } }
field < x	{field: {$lt: x } }
field >= x	{field: {$gte: x } }
field <= x	{field: {$lte: x } }
field <> x	{field: {$ne: x } }
not condition	{$not: { condition } }
field in (value)	{field: {$in: [value] } }
field like pattern	{field: {$regex: pattern} }
field between value and value2	{field: {$gte: value, $lte: value2 } }

Quando é necessário procurar por padrões em *string*, é importante relembrar que em MongoDB a procura é feita a partir da sintaxe *regex*, ou expressão regular, enquanto em SQL, é feita a partir dos *wildcards*. Para converter uma sintaxe, na outra, é necessário acrescentar ao inicio de cada padrão o caracter "^" e no final o caracter "$", depois disso, é substituído o caracter "%" por ".*?" e o caracter "_" por ".", como ilustrado na Tabela 5-8.

Tabela 5-8 Expressão SQL (Wildcards) equivalente à expressão Mongo (Regex)

Expressão SQL	Expressão Mongo
_	.
%	.*?
"_A%"	"^.A.*?$"

Se for necessário filtrar o conjunto de resultados com base em mais do que uma condição, MongoDB oferece a possibilidade de utilizar operadores *$and* e *$or*, como em SQL. Estes operadores são formados por uma lista, constituída pelo conjunto de condições pretendidas, como representado na Tabela 5-9.

Tabela 5-9 Expressão Mongo equivalente à expressão SQL, utilizando os operadores and e or

Expressão SQL	Expressão Mongo
condition and condition2	{$and: [{condition}, {condition2}] }
condition or condition2	{$or: [{condition}, {condition2}] }

5.4 Quarta Fase

A quarta e última fase de processamento da interrogação *select* está responsável por tratar as clausulas *group by*, *having*, *order by* e *limit*. O processamento de cada uma das clausulas está dependente de como a interrogação foi definida anteriormente, podendo modificar ou acrescentar parâmetros à mesma.

5.4.1 Group by

Se for pretendido agrupar o conjunto de resultados, com a utilização da clausula *group by*, o tradutor utilizará a fase *$group*, da framework *aggregate*. Contudo, em primeiro lugar, o sistema necessita de verificar como a interrogação Mongo foi contruída. Se não existir o parâmetro *$group*, é necessário acrescentá-lo à interrogação de agregação, com o campo "_id" preenchido com a ou as colunas desejadas pelo utilizador, no *group by*. Se já existir o parâmetro *$group*, por exemplo, devido à utilização de funções de agregação na segunda fase do processamento da interrogação, apenas será alterado o campo "_id" do valor *null* para os campos que representam as colunas pretendidas pelo utilizado, como ilustrado na Figura 5-5.

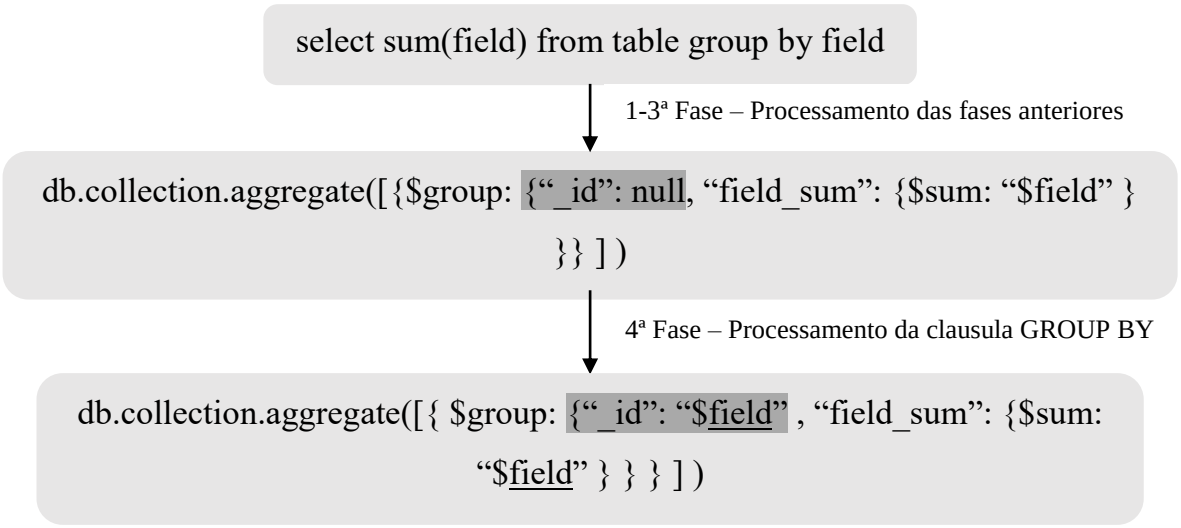

Figura 5-5 Processo da clausula Group by, tendo em conta que foram utilizadas funções de agregação (sum(field)), na segunda fase do processamento da interrogação

5.4.2 Having

Não existe uma transposição direta da clausula *having* para MongoDB. No entanto, quando é desejado filtrar o conjunto de resultados provenientes de funções de agregação, é necessário adicionar um parâmetro *$match*, com a condição desejada na clausula *having*, a seguir ao parâmetro $group, que especifica a função de agregação, como representado na Tabela 5-10. É importante mencionar que a condição no parâmetro $match tem que ser definida com o nome dado à função de agregação no parâmetro *$group*.

Tabela 5-10 Expressão Mongo equivalente à clausula Having

Expressão SQL	Expressão Mongo
having func(field) > x	{$group: {id: null, “field_func”: {$func: “$field ”}}}, {$match: {field_func: {$gt: x} } }

5.4.3 Order by

A clausula *order by* é equivalente ao método *sort* em MongoDB. Dependendo do tipo de interrogação, será utilizado a fase *$sort* ou o método *sort()*, como ilustrado na Tabela 5-11. Em ambos os casos, a ordenação ascendente é definida com o valor 1 e está associada a clausula *asc* em SQL, enquanto que, a ordenação descendente é definida com o valor 0 e está associada à clausula *desc*.

Tabela 5-11 Expressões Mongo equivalentes à função sort()

Expressão Find	Expressão Aggregate
find.({condition}, {fields}).sort({field : 1 })	aggregate.([{$sort: { field : -1 } }])

É importante mencionar que, sempre que for necessário ordenar o resultado de uma função de agregação, é obrigatório que a fase *$sort* esteja antes da fase *$group*, caso contrário, o resultado retornado não estará ordenado, Figura 5-6.

```
{$sort: {field : -1 } },  {$group: {_id: null, “field_fun”: {$fun: “$field ”} } }
```

Figura 5-6 A fase $sort deve vir primeiro que a fase $group, no pipeline

5.4.5 Limit

Por fim, o processamento da clausula *limit* é semelhante ao da clausula *order by*, ou seja, dependendo do tipo de interrogação Mongo, será utilizado a fase *$limit* ou o método *limit()*, Tabela 5-12. O sistema apenas aceita número inteiros, se o utilizador tentar introduzir um valor que não seja um número interior, surgirá uma mensagem a alertar o utilizador sobre o sucedido

Tabela 5-12 Expressões Mongo equivalentes à função limit()

Expressão Find	Expressão Aggregate
find.({condition}, {fields}).limit(x)	aggregate.([{$limit: x }])

Capítulo 6 – Estrutura do código e Manual de Utilizador

A estrutura do sistema é dividida em 3 partes:

1. Interface gráfica, utilizada como input/output do sistema
2. Transposição Mongo, utilizada para gerar o modelo relacional correspondente
3. Tradutor, utilizada para converter o *select* na interrogação mongo equivalente

Quando a aplicação é iniciada, surge uma janela, Figura 6-1, que permite ao utilizador optar por qual a ligação, cliente Mongo, pretendida. Antes de se ligarem, é obrigatório, iniciar o servidor *mongod*, (MongoDB 2018h). Este servidor é o principal processo, em segundo plano, do sistema MongoDB, e está responsável por gerir os pedidos feitos pelos utilizadores. O cliente Mongo, por omissão, está definido como "mongodb://localhost:27017/, esta ligação representa o servidor local. Se for pretendido alterá-lo, então, é necessário especificar a ligação com o formato uri, (MongoDB 2018d).

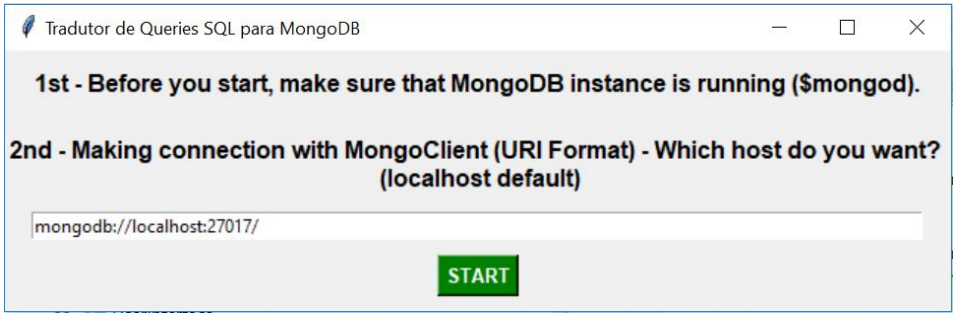

Figura 6-1 Primeira página do tradutor, serve para o utilizador escolher qual o cliente Mongo que pretende

Depois do servidor *mongod* estar a "correr" e o cliente Mongo estar definido, na próxima fase, questiona-se o utilizador, qual a coleção que deseja que seja transposta, Figura 6-2. Antes de eleger uma coleção, é necessário definir a base de dados onde essa coleção pertence. Os dois campos devem estar preenchidos e, caso, a base de dados ou a coleção, introduzida, não exista, o utilizador será alertado, do sucedido, Figura 6-3.

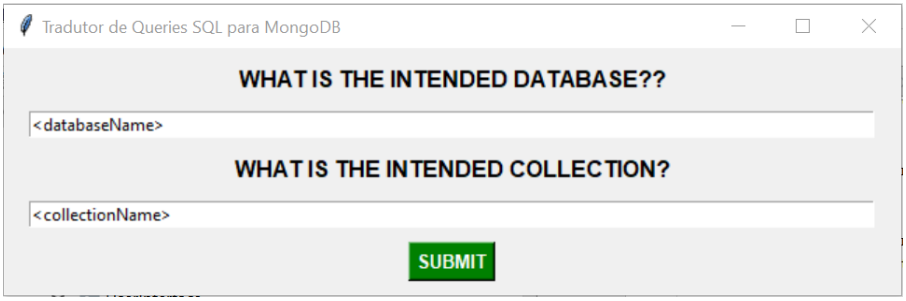

Figura 6-2 Janelas que permite ao utilizador escolher qual a coleção pretendida

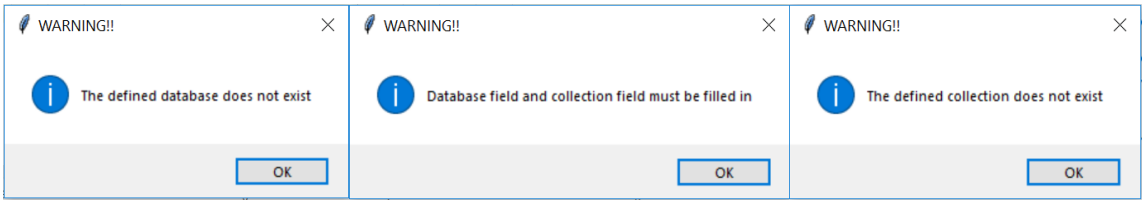

Figura 6-3 Mensagem de erro caso o utilizador não introduza a base de dados e a coleções, ou se introduzir umas que não existam

Após a coleção estar definida, a próxima parte que entra em ação é a Transposição Mongo, Figura 6-4. Esta parte é dividida por duas principais classes, a primeira classe chamada *Relation Section* e a segunda classe chamada *Relational Model*. A primeira classe, é a responsável por percorrer toda a coleção, definir o estado para cada campo, e criar um dicionário com a sintaxe "campo: estado". Enquanto que, a segunda classe, recebe como argumento, o dicionário criado na classe anterior, percorre toda coleção, e consoante o estado definido para cada campo, criará um novo dicionário composto pelas as tabelas que serão geradas. Este dicionário terá uma sintaxe "NomeTabela: {campo: TipoDeCampo, …}", isto significa que, será formado por uma chave, que define o nome da cada tabela, essa chave estará associada a outro dicionário, que será formado pelos múltiplos campos, que constituem a tabela, juntamente com o tipo de dados, por exemplo, *string* ou *int*.

Seguidamente, o modelo relacional passará por outras duas classes, a classe *DictToQuery* e a classe Dict*ToArray*. A classe *DictToQuery*, transformará o dicionário numa interrogação *Insert*, que será apresentada numa janela *popup*, como ilustrado na Figura 6-5. Permitindo ao utilizador, se desejar, introduzi-la noutra plataforma, com intuito de gerar um modelo relacional mais apelativo. A classe *DictToQuery*, transformará o dicionário numa estrutura *array*, que será utilizada, posteriormente, como auxiliar na tradução das interrogações.

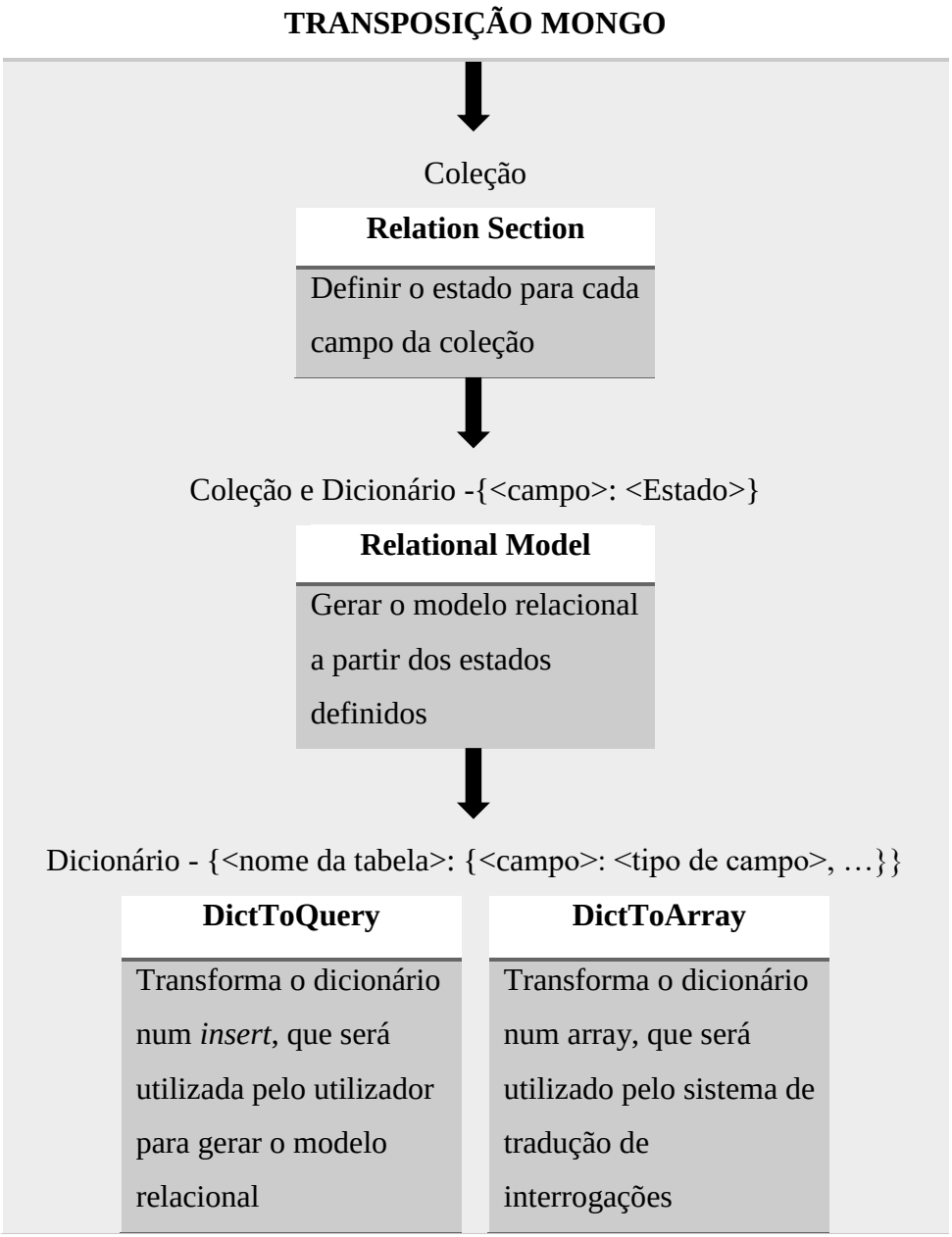

Figura 6-4 Processo de transposição para o modelo relacional, representado a partir das classes python definidas no programa

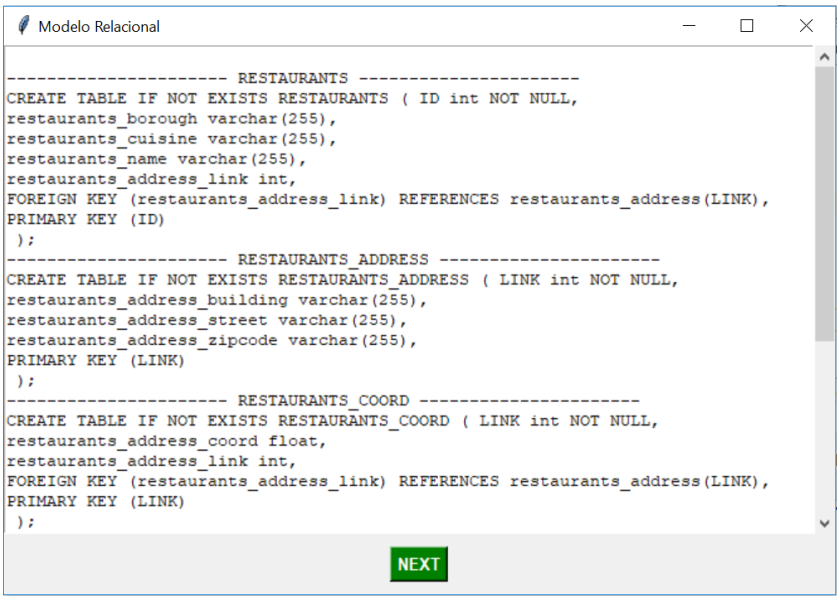

Figura 6-5 Janela que apresenta o comando insert correspondente ao modelo relacional transpostos

Depois do utilizador conhecer o modelo relacional correspondente, já pode interrogar a base de dados Mongo. Assim sendo, surgirá uma nova janela, com uma caixa de texto que permitirá o utilizador introduzir a interrogação *select* pretendida, Figura 6-6. Existe ainda a possibilidade de, caso o utilizador deseje mudar de coleção, utilizar o butão *return* para voltar atrás, para a janela representada na Figura 6-2.

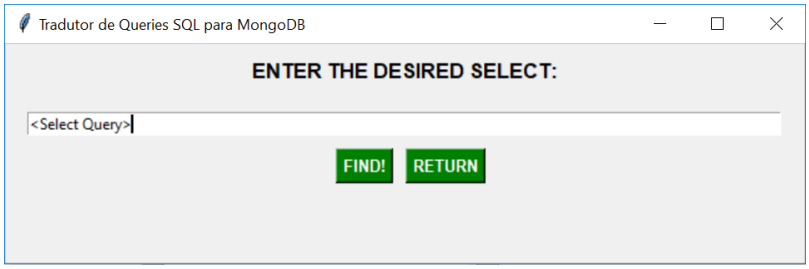

Figura 6-6 Janela que permite o utilizador introduzir a interrogação select pretendida

A tradução das interrogações, como já foi explicado nos capítulos anteriores, está dividida em quarto fases, Figura 6-7. Cada fase é representada por uma classe:

- A classe *collection Section*, que obtém a coleção correspondente à interrogação.
- A classe *fields section*, que obtém os campos pretendidos pelo utilizador.
- A classe *Condition Section*, que processa a condição associada à clausula *where*.
- A classe *statements Section*, que trata das clausulas *group by*¸having¸*order by*, *limit*

Após o sistema ter processado o *select*, todas as partes provenientes das diferentes fases, serão processadas numa classe chamada *querie section*. Esta classe é responsável por juntar todas as partes e criar a interrogação *find*, ou, caso, seja uma interrogação aggregate criar o pipeline ideal. Esta interrogação será utilizada para aceder aos dados na base de dados mongo, o resultado devolvido será retorna numa janela como ilustrado na Figura 6-8. Esta janela permite ao utilizador não só conhecer o conjunto de resultados, mas também conhecer a interrogação mongo gerada no tradutor. Neste caso a interrogação "db.restaurants.find({"borough": 'Brooklyn'}, {"name": 1, "_id": 0})" é o comando equivalente à interrogação "select restaurants_name from restaurants where restaurants_borough = 'Brooklyn' ".

É importante mencionar que o conjunto de resultados foi limitado a quinhentos documentos, isto porque, quando o resultado é muito grande, e é necessário apresentar cada documento na janela, o processo pode ficar lento, contudo, este valor é grande o suficiente para lidar com todas as situações comuns.

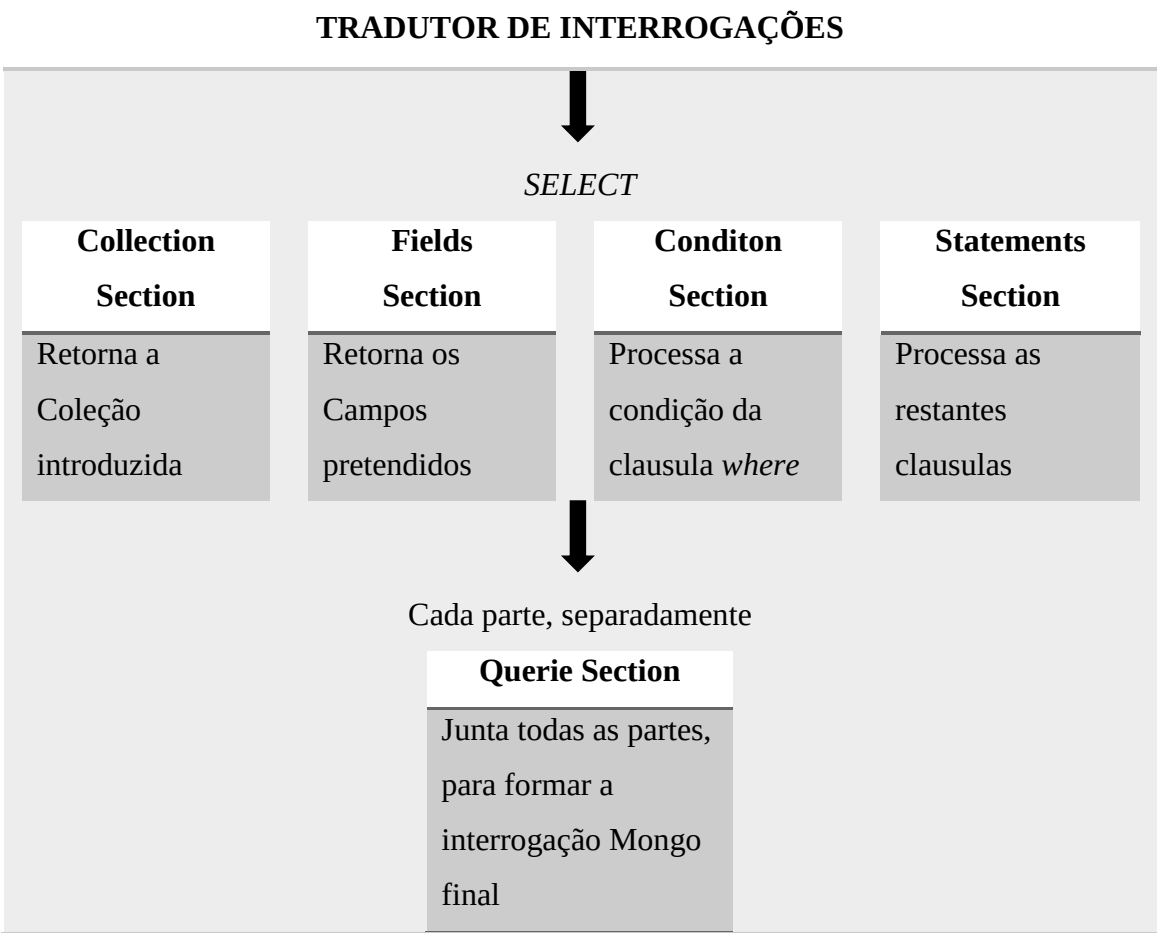

Figura 6-7 Processo de tradução das interrogações select, representado pelas classes python definidas no programa

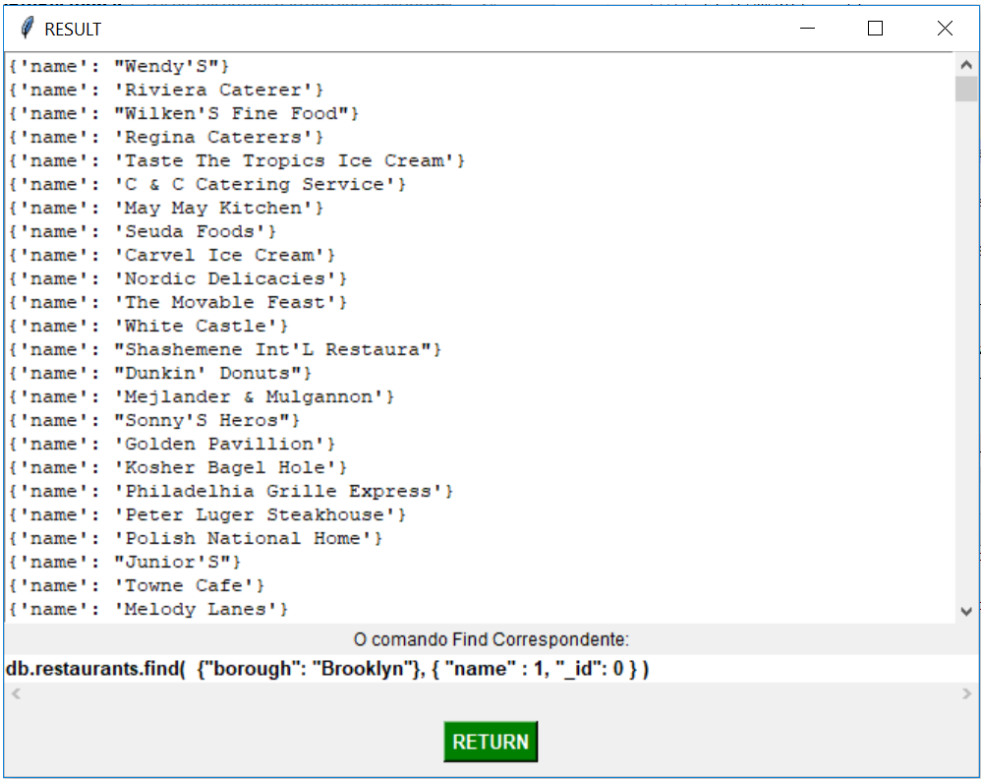

Figura 6-8 Janela que apresenta o conjunto de resultados, provenientes da interrogação select inserida

Capítulo 7 – Conclusão e Trabalhos Futuros

7.1 Conclusão

Com esta dissertação, conclui-se que é possível desenvolver um tradutor que converta as interrogações SQL em interrogações Mongo equivalentes, de modo a que a obtenção dos dados seja idêntica em ambos os casos. Ao longo dos anos, com o lançamento de novas versões, têm surgido múltiplas funcionalidades e operadores novos, que permitem trabalhar cada vez mais diversificadamente, incluindo desenvolver funcionalidades semelhantes à linguagem SQL. O maior desafio, nesta fase, foi conseguir fazer o processamento da interrogação *select*, sem que houvesse perca de informação, tendo em conta que, poderia ser constituída por diferentes expressões, múltiplas condições e que poderia ser introduzia pelo utilizador de diferentes maneiras.

Para desenvolver o sistema de transposição, foi possível definir um conjunto de regras de mapeamento entre o modelo de coleções, MongoDB, e o modelo relacional, sem que haja perda de informação. No entanto, esta parte veio a revelar-se bastante mais complicada do que estávamos à espera, devido à falta de esquema da estrutura Mongo. Cada documento poderia ter infinitos sub- níveis, que poderiam ter múltiplos documentos embebidos ou outros tipos de campos, tornando o documento umas vezes bastante simples, mas outras bastante mais complexas. Conseguir englobar todas estas situações, tornou-se um grande desafio, porque tudo que planeávamos tinha que ser o mais genérico possível. Sempre que começávamos uma parte, poderia surgir um novo problema, que não estávamos à espera e que poderia influenciar com outras partes já desenvolvidas. Demorou algum tempo a concluir esta fase, contudo, no final, penso que conseguimos desenvolver uma boa base para uma ferramenta de transposição.

Para terminar, apesar do tempo se ter tornado curto, penso que foi desenvolvido uma ferramenta bastante útil para os recentes utilizadores de MongoDB. Foi interessante conhecer esta base de dados, ao inicio foi um pouco confuso, porque ainda tinha o pensamento de um utilizador de SQL. Contudo, no final, acabo esta dissertação, a conhecer bastante melhor esta base de dados, a sua estrutura e principalmente o seu sistema de interrogações. Penso que MongoDB vai prevalecer no futuro, e que o tradutor desenvolvido, irá permitir aos

utilizadores habituados à linguagem SQL, aceder aos dados de uma base dados desconhecida, assim como, ter a oportunidade de conhecer os comandos mongo equivalentes, podendo ser uma mais valia para quem quiser entrar neste mundo.

7.2 Trabalhos Futuros

Como trabalho futuro, fica pendente, melhorar a performance da transposição para o modelo relacional. Em coleções muito grandes, a transposição pode demorar alguns minutos, podendo não ser apelativo para o utilizador. A melhoria da performance passa, não só, por tentar utilizar, sempre que possível, bibliotecas já existentes em Python, tentar evitar ao máximo ciclos *for*, a utilização de *appends*, ou o processamento desnecessário de dados.

Para além do melhoramento da performance, outra melhoria será acrescentar ao sistema de transposição, os documentos que referenciam outros documentos em diferentes coleções. Esta funcionalidade permitirá que seja possível transpor, de uma só vez, todas as coleções que compõem a base de dados. Estaria, em paralelo com o processamento dos estados de cada campo, verificando todos os outros campos, se, em primeiro lugar, eram valores do tipo *ObjectID*, *string* ou *int*. Se fosse, então seriam guardados numa estrutura para, posteriormente, quando se processasse as outras coleções, comparar com o campo _id de cada documento. Se os valores coincidissem, significava que o campo era uma referência. Nesta situação, seria gerada, obrigatoriamente, uma nova tabela, pois a relação entre os campos nunca poderia ser Um-Um. Tal se sucede, porque, quando estamos a falar deste tipo de relação, onde ambos os campos vêm sempre juntos um com o outro, assumindo que a base de dados está bem estruturada, não haveria nenhum partido positivo em fazer referências em vez de embeber os documentos.

Nas interrogações, fica por desenvolver a tradução da declaração *distinct*, dos *joins* entre tabelas de diferentes coleções e de sub-queries. A mais acessível é o processamento da declaração *distinct*. Em MongoDB, já existe uma função chamada *distinct*, que tem a mesma funcionalidade, porém, é necessário ter em atenção quando é pretendido aceder a campos do tipo array. Nesta situação, se queremos definir uma condição, é importante utilizar a função *$elemMatch*, explicada no capítulo Estado de Arte. Em relação às *sub-queries*, MongoDB não permite utilizar uma interrogação dentro de outra, contudo, é possível criar uma variável proveniente de uma interrogação, e utilizar essa variável noutra interrogação. A ligação entre

dados de diferentes coleções, deve ser feita a partir da fase *$lookup*, da framework *Aggregate*. Esta fase é como um *left join*, em SQL, e permite relacionar diferentes coleções na mesma base de dados.

No final, após as anteriores funcionalidades estarem implementadas, desenvolver um sistema que alerte o utilizador sempre que seja introduzida uma interrogação mal estruturada, por exemplo, com as clausulas mal escritas ou ordenadas erradamente.

Referências

"Apache Drill." https://drill.apache.org/ (January 1, 2018).

Brewer, E. 2012. "CAP Twelve Years Later: How the 'Rules' Have Changed." *Computer* 45(2): 23–29. http://ieeexplore.ieee.org/document/6133253/.

Brewer, Eric. 2000. "Towards Robust Distributed System." *Symposium on Principles of Distributed Computing (PODC)*. http://www.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

Cattell, Rick. 2011. "Scalable SQL and NoSQL Data Stores." *ACM SIGMOD Record* 39(4): 12. http://portal.acm.org/citation.cfm?doid=1978915.1978919.

Fowler, Adam. 2015. *NoSQL For Dummies*. John Wiley & Sons, Inc.

Gilbert, Seth, and Nancy Lynch. 2005. "Brewer ' s Conjecture and the Feasibility of Consistent , Available , Partition-Tolerant Web Services." : 51–59.

Gilbert, Seth, and Nancy A Lynch. 2014. "Perspectives on the CAP Theorem Accessed Detailed Terms Perspectives on the CAP Theorem." : 0–10.

Hecht, Robin, and Stefan Jablonski. 2011. "NoSQL Evaluation: A Use Case Oriented Survey." *Proceedings - 2011 International Conference on Cloud and Service Computing, CSC 2011* (December): 336–41.

Hoi, Sunny. 2017. "JSON vs XML: Which Format To Use For Your API?" https://www.sunnyhoi.com/json-vs-xml-format-use-api/ (January 1, 2018).

Houlihan, Rick. 2014. "The Aggregation Framework." *MongoDB*. https://www.mongodb.com/presentations/aggregation-framework-0 (September 18, 2018).

Hows, David, Eelco Plugge, Peter Membrey, and Tim Hawkins. 2013. *The Definitive Guide to MongoDB*. Second Edi. Apress.

Kaur, Karamjit, and Rinkle Rani. 2013. "Modeling and Querying Data in NoSQL Databases." *Proceedings - 2013 IEEE International Conference on Big Data, Big Data*

2013: 1–7.

Khan, Sanobar, and Vanita Mane. 2013. “SQL Support over MongoDB Using Metadata.” *International Journal of Scientific and Research Publications* 3(10): 2250–3153. www.ijsrp.org.

“Klaus.Dk.” 2012. https://klaus.dk/sqltomongodb/ (January 1, 2018).

Leavitt, Neal. 2010. “Will NoSQL Databases Live Up to Their Promise?” *Computer* 43(2): 12–14. http://ieeexplore.ieee.org/document/5410700/.

McCreary, Dan, and Ann Kelly. 2014. *Manning Sense of NoSQL*. Manning Publications Co.

MongoDB, Inc. 2018a. “MongoDB Manual.” https://docs.mongodb.com/manual/ (September 24, 2018).

MongoDB, Inc. 2018b. “MongoDB Manual - $and.” https://docs.mongodb.com/manual/reference/operator/query/and/ (September 18, 2018).

MongoDB, Inc. 2018c. “MongoDB Manual - $or.” https://docs.mongodb.com/manual/reference/operator/query/or/ (September 18, 2018).

MongoDB, Inc. 2018d. “MongoDB Manual - Connection String URI Format.” https://docs.mongodb.com/manual/reference/connection-string/.

MongoDB, Inc. 2018e. “MongoDB Manual - Data Storages.” https://docs.mongodb.com/manual/core/data-modeling-introduction/ (September 24, 2018).

MongoDB, Inc. 2018f. “MongoDB Manual - Db.Collection.Find().” https://docs.mongodb.com/manual/reference/method/db.collection.find/ (September 19, 2018).

MongoDB, Inc. 2018g. “MongoDB Manual - Indexes.” https://docs.mongodb.com/manual/indexes/.

MongoDB, Inc. 2018h. “MongoDB Manual - Mongod.” https://docs.mongodb.com/manual/reference/program/mongod/ (September 24, 2018).

"NoSQL Booster." https://nosqlbooster.com/ (September 18, 2018).

Padhy, Rabi Prasad, Manas Ranjan Patra, and Suresh Chandra Satapathy. 2011. "RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database's." *International Journal of Advanced Engineering Sciences and Technologies* 11(11): 15–30.

Pritchett, Dan. 2008. "Base: An Acid Alternative." *Queue* 6(3): 48–55. http://portal.acm.org/citation.cfm?doid=1394127.1394128.

"QueryMongo." *Stitch*. http://www.querymongo.com/ (August 20, 2001).

"SlamData." 2017. https://slamdata.com/ (January 1, 2018).

"Studio3T." https://studio3t.com/features/sql-query/ (January 1, 2018).

Vaish, Gaurav. 2013. Pack Publishing Ltd *Getting Started with NoSQL*.

Wodehow, Carey. "SQL vs. NoSQL Databases: What's the Difference?" *UpWork*. https://www.upwork.com/hiring/data/sql-vs-nosql-databases-whats-the-difference/ (January 1, 2018).