SEREIA – Busca por Palavras-Chave em Document Stores

Ariel Afonso[1], **Paulo Martins**[1], **Altigran da Silva**[1]

[1]Instituto de Computação – Universidade Federal do Amazonas (UFAM)
69.077-000 – Manaus – AM – Brasil

{ariel.afonso,paulo.martins,alti}@icomp.ufam.edu.br

Resumo. *Gerenciadores de documentos (GDs) ou* document stores*, como MongoDB e CouchDB, têm se tornado cada vez mais populares devido à flexibilidade em carregar e recuperar dados em larga escala usando documentos semi-estruturados, pois evitam a necessidade de definição de esquemas antes da ingestão de dados. Por outro lado, especificar consultas neste tipo de sistema é uma tarefa complexa, ainda mais que em sistemas relacionais, devido à natureza semi-estruturada dos documentos e à possibilidade de aninhar itens de dados complexos. Para lidar com esse problema, apresentamos uma abordagem chamada* SEREIA[1]*, que permite a execução de consultas por palavras-chave sobre coleções de documentos armazenados em GDs sem necessidade de conhecimento da estrutura das coleções. Nossa abordagem é baseada na geração de* Candidate Join Networks *que representam diferentes interpretações da consulta fornecida, a fim de ranqueá-las e escolher a alternativa mais adequada. Experimentos realizados em um banco de dados representativo, contendo diversas coleções de documentos, mostram que nossa abordagem foi eficaz em gerar consultas estruturadas que satisfazem a intenção original do usuário expressa na consulta por palavras-chave, alcançando valores de Precisão e MRR de 1.0 e 0.98, respectivamente.*

1. Introdução

A adoção de gerenciadores de documentos (GDs) ou *document stores*, como MongoDB e CouchDB, aumentou significativamente nos últimos anos devido à flexibilidade em carregar, armazenar e recuperar dados em larga escala disponíveis em documentos semi-estruturados [DiScala and Abadi 2016, Tahara et al. 2014]. Em particular, essa facilidade provém da abordagem *schema later* adotada por essa classe de sistemas, que possibilita o aumento da produtividade de desenvolvedores, principalmente em ambientes ágeis [Liu et al. 2014], ao permitir a ingestão de documentos pelos sistemas sem a necessidade de reforçar a conformidade de estrutura entre os documentos, diferentemente da abordagem adotada em BDs relacionais [Tahara et al. 2014]. Assim, GDs são empregados em muitos domínios de aplicação diferentes [Sadalage and Fowler 2012].

Um importante efeito colateral dessa flexibilidade, é a possibilidade de criar documentos semi-estruturados, com hierarquias de aninhamento complexas [DiScala and Abadi 2016]. Isso é feito sem a necessidade de definição de esquemas, ao contrário dos sistemas relacionais [Chasseur et al. 2013]. Dessa forma, a realização de consultas sobre GDs pode ser ainda mais complexa do que em sistemas relacionais, pois exige o conhecimento dos detalhes da estruturação dos documentos. Essa complexidade é exacerbada quando a consulta necessita de junções entre documentos de coleções

[1]Este nome alude a ser esse um sistema hibrido de recuperação de informação e bancos de dados.

distintas. Outro efeito colateral importante é que as linguagens de consulta tipicamente utilizadas em GDs apresentam recursos para dar suporte adequado aos documentos semi-estruturados, o que muitas vezes implica em aumento de complexidade na escrita de consultas em comparação com SQL.

Além de linguagens de consulta nativas, abordagens alternativas para consultas em GDs têm sido propostas na literatura. Sistemas como *Sinew* [Tahara et al. 2014] e *Argo* [Chasseur et al. 2013] adotam uma camada intermediária que é responsável por mapear os documentos para tuplas de um BD relacional, de forma que as consultas possam ser feitas em SQL convencional. Outra abordagem permite o uso de SQL em documentos sem a necessidade de mapear os elementos para um BD relacional [Liu et al. 2016]. Apesar de estas abordagens facilitarem consultas em GDs, ainda é necessário que o usuário tenha conhecimento das estruturas das coleções a serem consultadas.

Neste artigo propomos o *SEREIA*, um sistema que permite que consultas não-estruturadas baseadas em palavras-chave, típicas de sistemas de recuperação de informação, sejam realizadas sobre coleções de documentos semi-estruturados armazenados em GDs, permitindo inclusive a recuperação de resultados que incluem junções entre documentos de coleções distintas. Especificamente, o *SEREIA* tem como objetivo gerar uma consulta estruturada na linguagem nativa do GD correspondente à consulta original não-estruturada fornecida pelo usuário. Isso evita que o usuário tenha que conhecer os detalhes da estruturação dos documentos e da organização das coleções, além da sintaxe e recursos da linguagem nativa de consulta.

Embora existam diversas abordagens na literatura para consultas por palavras-chave sobre BDs relacionais, nós utilizamos como base para o nosso trabalho o sistema MatCNGen [Oliveira et al. 2015, Oliveira et al. 2018, Oliveira et al. 2020], que representa o estado da arte nessa classe de sistemas. Pelo levantamento que realizamos na literatura recente, o *SEREIA* é o primeiro sistema proposto para a tarefa de consulta por palavras-chave em GDs. De forma similar ao MatCNGen, o *SEREIA* inicialmente mapeia cada palavra-chave da consulta para atributos dos documentos, com base no nome do atributo ou nos seus valores. O sistema então combina esses mapeamentos, de forma que cada combinação corresponde a uma possível interpretação da consulta fornecida. Um algoritmo de ranqueamento é então usado para determinar que combinações têm a maior probabilidade de satisfazer essa consulta. A partir das melhores combinações, o sistema então analisa maneiras de realizar junções entre mapeamentos que as compõem. O resultado dessa análise é usado, ao final do processo, para gerar uma consulta estruturada correspondente à consulta original na linguagem nativa do GD.

Para validar nossa proposta, conduzimos experimentos usando o MongoDB[2], que é um document store bastante popular e conhecido[3]. Como base para os experimentos, utilizamos um BD de documentos no formato *JSON* disponibilizado pelo conhecido site de avaliações Yelp!. Esse BD consiste em várias coleções distintas de documentos ricos em dados, todos reais, gerados por usuários. Para cada consulta de uma série de consultas por palavras-chave para este BD, analisamos os resultados e o processo de geração executado pelo *SEREIA*, usando métricas de recuperação de informação como *Mean Reciprocal Ranking* (MRR) e *Precision at position K* (P@K) [Baeza-Yates and Ribeiro-Neto 2011].

[2] `https://www.mongodb.com/`

[3] `https://db-engines.com/en/ranking/document+store`

Além disso, medimos também o tempo gasto pelo sistema no processo de geração das consultas. Os resultados obtidos demonstram que o sistema é útil em auxiliar usuários a consultar dados em GDs sem conhecimento da estrutura, da sintaxe de consulta e de como reunir informações para gerar a resposta desejada. Note-se que no presente artigo nos limitamos a realizar experimentos em um único GD e conjunto de consultas, pois nosso objetivo é apresentar o *SEREIA* e mostrar como as técnicas de recuperação de informação usadas em algoritmos já existentes na literatura podem ser adaptadas para este contexto.

Este artigo está organizado da seguinte forma. A Seção 2 apresenta o contexto deste trabalho assim como trabalhos relacionados. A Seção 3 apresenta uma visão geral do sistema proposto e a Seção 4 discute seus detalhes. A Seção 5 apresenta resultados experimentais. Por último, a Seção 6 apresenta conclusões e próximos passos.

2. Contexto e Trabalhos Relacionados

Sistemas para busca por palavras-chave em BDs vem sendo estudados há quase duas décadas [Hristidis and Papakonstantinou 2002]. De forma geral, existem duas categorias de sistemas. Na primeira categoria, os sistemas *baseados em esquema* têm por objetivo gerar uma consulta estruturada em SQL a partir de uma consulta por palavra-chave. Estes sistemas constroem um grafo, chamado *Schema Graph*, a partir do esquema do banco de dados, onde os nós representam as relações do BD e as arestas representam chaves estrangeiras Deste grafo são extraídas uma ou mais *Candidate Join Networks* (CJNs), que são árvores cujo conjunto de nós representa uma possível interpretação da consulta original e cujas arestas indicam que junções devem ser realizadas para gerar o resultado esperado. De cada CJN pode ser gerada uma consulta SQL. Os sistemas do estado da arte nessa abordagem procuram determinar de antemão quais as CJNs que melhor representam a consulta original, melhorando a qualidade e o desempenho do processo [Oliveira et al. 2015, Oliveira et al. 2018, Oliveira et al. 2020]. Na segunda categoria, os sistemas *baseados em instância* focam em materializar tuplas em um grafo, o *Data Graph*, onde os nós representam as tuplas do BD contendo as palavras-chave, enquanto as arestas representam referências de chave estrangeira. Com isso, a resposta corresponde a sub-árvores desse grafo que minimizam a distância entre os nós que apresentam as palavras-chaves. O *SEREIA* é um sistema baseado em esquema desenvolvido a partir das ideias propostas no sistema MatCNGen [Oliveira et al. 2020]. Apesar de GDs não possuírem nativamente restrições de integridade referencial como em BDs relacionais, documentos de uma coleção podem apresentar referências a documentos de outras coleções. Assim, nossa abordagem oferece a possibilidade de consulta envolvendo diferentes coleções ao gerar as junções entre os documentos com base nos atributos de referência desses documentos.

Recentemente, sistemas para consultas sobre BDs utilizando linguagem natural têm ganhado interesse. Estes se dividem em duas vertentes: *Sistemas Centrados em Dados (SCD)* e *Sistemas Centrados em Linguagem (SCL)*. Sistemas do primeiro tipo, como NaLIR [Li and Jagadish 2014] focam em mapear elementos da consulta para elementos do esquema do BD usando recursos genéricos como regras de dependência sintáticas e ontologias. Sistemas do segundo tipo, como SQLizer [Yaghmazadeh et al. 2017] focam em traduzir as consultas em linguagem natural para SQL usando aprendizagem profunda. Embora interessantes e promissores, estes sistemas baseados em linguagem natural destinam a consultas mais elaboradas, onde o usuário pode especificar, por exemplo,

agregações. Sistemas como o *SEREIA* tem como principal vantagem a simplicidade de recuperar informação de BDs somente com algumas palavras-chave.

Alguns trabalhos na literatura buscam facilitar consultas sobre *document stores* através do mapeamento de documentos para estruturas relacionais, beneficiando-se do conhecimento de usuários em SQL. Sistemas como *Argo* [Chasseur et al. 2013] e *Sinew* [Tahara et al. 2014] proveem uma camada intermediária que fornece a possibilidade de consultar dados presentes em documentos semi-estruturados através do uso de tabelas relacionais. Para isso, os dados carregados passam por processos de mapeamento e materialização em tabelas relacionais no momento da ingestão para que possam ser acessados posteriormente. Outras abordagens [Liu et al. 2016] utilizam um mapeamento nativo fornecido pelo BD que consulta os documentos sem que haja mapeamento para uma estrutura relacional. Apesar destas abordagens facilitarem consultar dados com a linguagem SQL, o desafio de conhecer a estrutura subjacente das coleções de documentos permanece. Para atacar esse problema, nosso sistema permite que o usuário realize consultas por palavras-chave sobre *document stores* sem o conhecimento prévio da estrutura dos documentos.

3. Visão Geral

Nesta seção, apresentamos a visão geral do *SEREIA* para consultas por palavras-chave em document stores. Para a base da nossa discussão, usamos como exemplo o BD Yelp!. Na Figura 1 ilustramos a organização de um fragmento desse BD com algumas de suas coleções de documentos e partes de alguns desses documentos. Também ilustramos como estas coleções se relacionam. Os exemplos de documentos e consultas apresentados seguem a sintaxe do sistema MongoDB, usado também em nossos experimentos. A seguir, exemplificamos o processo de geração de consultas para o MongoDB dada uma consulta baseada em palavras-chave fornecida por um usuário.

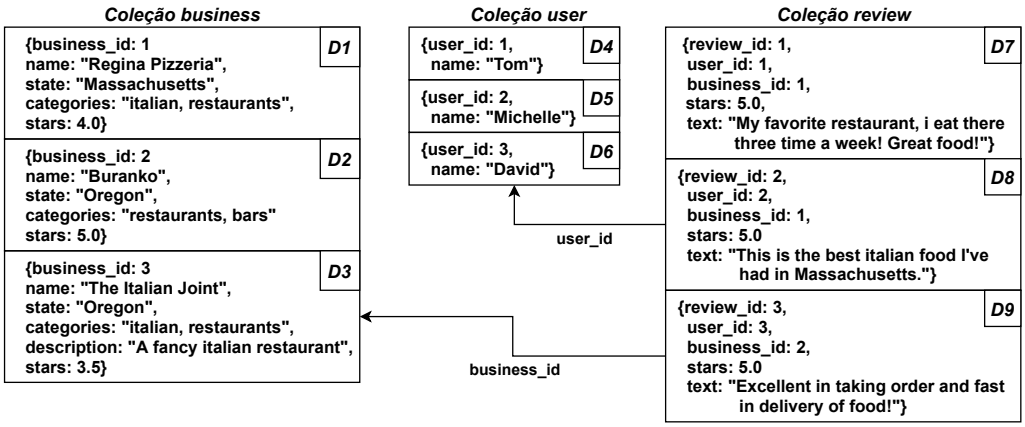


Figura 1. Organização de um fragmento no BD Yelp!

Considere que um usuário fornece a consulta por palavras-chave "italian restaurants Massachusetts reviewed michelle", na qual a intenção é recuperar todos os restaurantes italianos no estado de Massachusetts que possuem uma avaliação pela usuária *michelle*. Dada a Figura 1, vemos que o termo "*Massachusetts*" possui maiores chances de estar associado ao atributo *state* da coleção `BUSINESS`. Por outro lado, vemos que os termos "*italian*" e "*restaurants*" possuem maiores chances de estarem associados ao atributo *categories* pertencente à coleção `BUSINESS`. No caso de GDs, nota-se que atributos

podem ou não ser encontrados dentro de documentos da mesma coleção, como o atributo *description* da coleção BUSINESS.

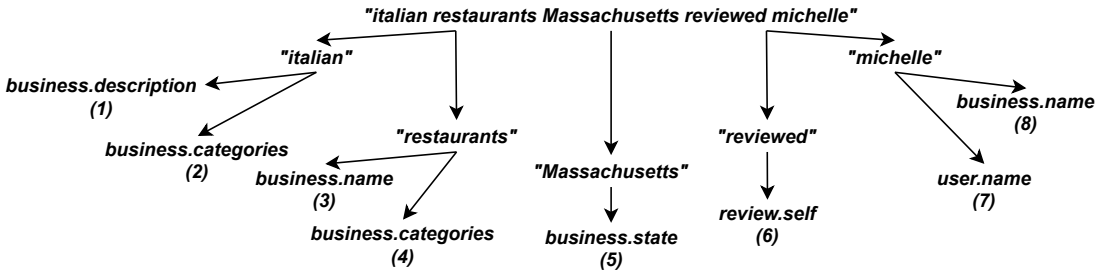


Figura 2. Consulta por palavras-chave e mapeamentos para elementos do BD.

Ilustramos, na Figura 2, para cada palavra-chave, algumas das possibilidades de mapeamentos. Usamos a notação de ponto, onde representamos a coleção seguida por um de seus atributos. A partir do conjunto de mapeamentos para cada palavra-chave, podemos combinar os termos para gerar possíveis representações da consulta. Através da numeração dos mapeamentos explicitada na Figura 2, podemos exemplificar algumas combinações, como: $\{1, 3, 5, 6, 7\}$, $\{2, 4, 5, 6, 7\}$, etc. Essas combinações serão usadas para definir as condições de associação de documentos de coleções.

```
db.user.aggregate(
    [{"$match": { "$expr": { "$regexMatch": {
        "input": "$name", "options": "i", "regex": "michelle"}}}},
     {"$lookup": {
        "as": "review", "foreignField": "user_id",
        "from": "review", "localField": "user_id"}},
     {"$lookup": {
        "as": "business", "foreignField": "business_id",
        "from": "business", "localField": "review.business_id"}},
    {"$addFields": {
        "business_filtered": {
            "$filter": { "as": "field", "cond": { "$and": [
                    {"$regexMatch": {
                        "input": "$$field.categories", "options": "i",
                        "regex": "italian"}},
                    {"$regexMatch": {
                        "input": "$$field.categories", "options": "i",
                        "regex": "restaurants"}},
                    {"$regexMatch": {
                        "input": "$$field.state", "options": "i",
                        "regex": "massachusetts"}}]},
                "input": "$business"}}}}
    {"$project": {"business": 0}}])
```

Figura 3. Consulta estruturada usando a sintaxe do MongoDB.

Dadas as combinações, o *SEREIA* gera consultas que podem ser executadas sobre o MongoDB, como ilustra a Figura 3 que corresponde à combinação $\{1, 3, 5, 6, 7\}$. A consulta Q_1 recupera os documentos da coleção BUSINESS que apresentam avaliações da usuária *michelle* para restaurantes italianos localizados em Massachusetts, satisfazendo a intenção do usuário. De forma geral, podem haver diversas alternativas de consultas para o MongoDB dada uma consulta por palavras-chave. O *SEREIA* é responsável por estimar a alternativa mais adequada para satisfazer a intenção do usuário por meio de um ranqueamento das combinações geradas. Na próxima seção, descrevemos as etapas de execução do *SEREIA* e suas estruturas, apresentando uma breve discussão de cada etapa.

4. *SEREIA*

Nesta seção, apresentamos a arquitetura e o funcionamento do *SEREIA*. A Figura 4 apresenta as etapas da geração de consultas para o MongoDB. O processo possui 3 etapas, as quais são detalhadas a seguir.

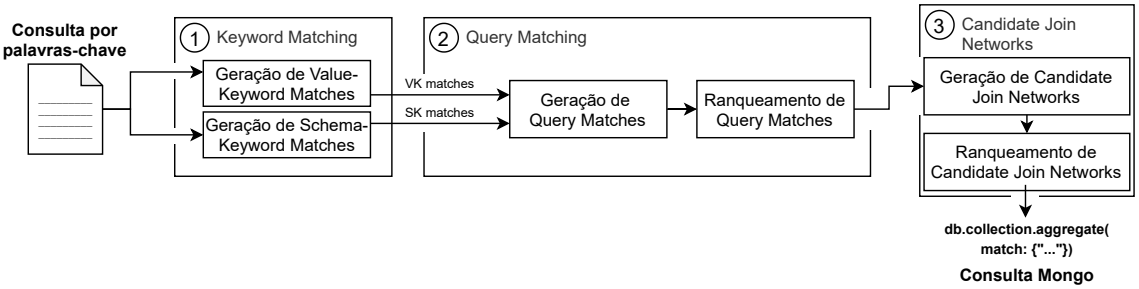


Figura 4. Arquitetura Geral do *SEREIA*

4.1. *Keyword Matching*

O processo inicia quando uma consulta composta por palavras-chave é fornecida pelo usuário e o *SEREIA* tenta associar cada palavra-chave a um atributo dos documentos de uma coleção. Isso pode ser feito com base no nome do atributo ou nos seus valores, como descrito abaixo. Seja Q uma consulta por palavras-chave, D uma coleção de documentos, e $\mathcal{A} = \{A_1, \ldots, A_n\}$ o conjunto dos atributos usados nos documentos de D.

Um ***value-keyword match***[4] (VKM) é um subconjunto (*subcoleção*) de D cujos documentos contém um subconjunto das palavras de Q em pelo menos um de seus atributos. Mais precisamente, um VKM é um conjunto de documentos $D^V[A_1^{K_1}, \ldots, A_n^{K_n}] = \{d \mid d{\in}D \wedge \forall A_i : d[A_i] \cap Q = K_i\}$, onde $d[A_i]$ é o conjunto de palavras no atributo A_i de d e cada K_i é um subconjunto de Q disjunto dos demais. Para simplificar a notação, omitimos conjuntos vazios de palavras-chave. Por exemplo, usamos a notação $D^V[A_1^{K_1}]$ para representar $D^V[A_1^{K_1}, \ldots, A_n^{K_n}]$, se todos $K_2, \ldots, K_n$ são vazios.

Considerando a consulta Q ="*italian restaurants Massachusetts reviewed michelle*" e as coleções de documentos ilustrados na Figura 1, alguns possíveis VKM são: $\texttt{USER}^V[name^{\{michelle\}}]$={$D5$}, $\texttt{BUSINESS}^V[categories^{\{italian,restaurants\}}]$={$D1, D3$} e $\texttt{BUSINESS}^V[state^{\{Massachusetts\}}]$={$D1$}. Para a mesma consulta, no banco de dados que utilizamos em nossos experimentos são gerados 28 VKMs para essa consulta.

Um ***schema-keyword match*** (SKM) é um subconjunto (*subcoleção*) de D cujos documentos contêm ao menos um atributo cujo nome corresponde (*matches*) a pelo menos uma palavra da consulta Q. Mais precisamente, um SKM é um conjunto de documentos $D^S[A_1^{K_1}, \ldots, A_n^{K_n}] = \{d \mid d{\in}D \wedge \forall A_i \in \mathcal{A}(d) \wedge \mathit{match}(A_i, K_i)\}$ onde $\mathcal{A}(d)$ é o conjunto de atributos do documento d e *match*(A_i, K_i) é verdadeiro se o nome do atributo A_i corresponde ao subconjunto K_i da consulta, sendo K_i disjunto dos demais. Para tratar o caso em que a consulta faz referência ao nome da coleção, ao invés de a um atributo específico, estendemos o conjunto $\mathcal{A}(d)$ adicionando um atributo chamado *self* e consideramos que *match*(*self*,K_i) é verdadeiro se K_i corresponde ao nome da coleção. A função *match* pode ser implementada de diversas maneiras, usando métricas de similaridade entre strings como Levenshtein, Soft-TF-IDF e distância entre vetores que representam palavras. Em nossa atual implementação utilizamos duas métricas baseadas no banco de dados léxico *WordNet*[5]: a similaridade *Path* e a similaridade *Wu-Palmer*, já utilizadas em trabalhos anteriores [Li and Jagadish 2014, Bhalotia et al. 2002]. O detalhamento destas funções será omitido neste artigo por questões de espaço, sem comprometer o entendimento do leitor,

[4]O conceito de *Value-keyword matches* generaliza o conceito de *tuple-sets* descritos na literatura [Hristidis and Papakonstantinou 2002, Oliveira et al. 2020].

[5]`https://wordnet.princeton.edu/`

nem nossos resultados e conclusões. Note-se que em caso de atributos aninhados em um documento estruturado, a função *match* precisa considerar também todos os nomes em um caminho na hierarquia.

Seja Q =*"italian restaurants 5 description"* e as coleções da Figura 1. Apenas dois SKMs são gerados: $\texttt{BUSINESS}^V[description^{\{description\}}]=\{D3\}$ e $\texttt{BUSINESS}^V[self^{\{restaurants\}}]=\{D1, D2, D3\}$. Observe-se que, nesse caso, a palavra-chave *"restaurants"* foi mapeada para a coleção BUSINESS pela aplicação da função *match*.

4.2. *Query Matching*

O objetivo desta etapa é combinar VKMs e SKMs para gerar *Query Matches* ou *QMs*, isto é, combinações válidas de ocorrências das palavras das consultas nos elementos do BD. Considere a consulta *"italian restaurants Massachusetts reviewed michelle"*. Apresentamos abaixo alguns combinações possíveis destes KMs, onde cada palavra da consulta está presente em pelo menos um dos KMs gerados. Para a mesma consulta, no BD que utilizamos em nossos experimentos, são gerados ao todo 263 combinações a partir das combinações dos VKMs e SKMs apresentados. Algumas delas são apresentadas abaixo.

C_1 = $\{\texttt{USER}^V[name^{\{michelle\}}],$
$\texttt{BUSINESS}^V[categories^{\{restaurants,italian\}}, state^{\{Massachusetts\}}], \texttt{REVIEW}^S[self^{\{reviewed\}}]\}$
C_2 = $\{\texttt{USER}^V[name^{\{michelle\}}], \texttt{BUSINESS}^S[self^{\{restaurants\}}]^V[name^{\{italian\}}, state^{\{Massachusetts\}}],$
$\texttt{REVIEW}^S[self^{\{reviewed\}}]\}$
C_3 = $\{\texttt{USER}^V[name^{\{michelle\}}], \texttt{BUSINESS}^V[categories^{\{restaurants,italian\}}],$
$\texttt{BUSINESS}^V[name^{\{italian\}}], \texttt{REVIEW}^S[self^{\{reviewed\},}]\}$
C_4 = $\{\texttt{USER}^V[name^{\{michelle\}}], \texttt{BUSINESS}^V[categories^{\{restaurants,italian\}}],$
$\texttt{BUSINESS}^V[name^{\{italian\}}], \texttt{BUSINESS}^V[state^{\{Massachusetts\}}], \texttt{REVIEW}^S[self^{\{reviewed\}}]\}$

Em princípio, qualquer combinação das KMs de uma consulta atende ao requisito de combinar conjuntos de documentos, que quando corretamente conectados, satisfazem a consulta. Por exemplo, a combinação C_1 recupera os documentos D_1 da coleção BUSINESS, D_5 da coleção USER e D_8 da coleção REVIEW, os quais satisfazem a consulta. No entanto, em nosso trabalho, em consonância com a semântica adotada em outros sistemas de palavras-chave sobre BDs [Hristidis and Papakonstantinou 2002, Oliveira et al. 2020], consideramos apenas as combinações que apresentam uma cobertura *total* e *minimal* sobre a consulta. A cobertura total garante que todas as palavras da consulta estarão presentes na resposta. A cobertura minimal garante que não haverão documentos com informação redundantes na resposta. As combinações de KMs que são totais e minimais são chamadas de *Query Matches*. No exemplo acima, apenas as combinações C_1 e C_2 são consideradas Query Matches. A combinação C_3, que não inclui todas as palavras da consulta, não é total, enquanto a consulta C_4 não é minimal porque inclui palavras da consulta de forma redundante. Formalizamos abaixo esse conceito.

Sejam VK e SK, respectivamente os VKMs e SKMs de uma consulta por palavras-chave Q sobre um banco de dados D em uma *document store*. Considere um subconjunto M de $VK \cup SK$ da forma:

$$M = \{D_1^S[A_{1,1}^{K_{1,1}^S}, \ldots, A_{1,m_1}^{K_{1,m_1}^S}]^V[A_{1,1}^{K_{1,1}^V}, \ldots, A_{1,m_1}^{K_{1,m_1}^V}], \ldots, D_n^S[A_{n,1}^{K_{n,1}^S}, \ldots, A_{n,m_n}^{K_{n,m_n}^S}]^V[A_{n,1}^{K_{n,1}^V}, \ldots, A_{n,m_n}^{K_{n,m_n}^V}]\}$$

Considere também $C = \bigcup_{\substack{1\leq i\leq n \\ 1\leq j\leq m_i \\ X\in\{S,V\}}} K_{i,j}^X$, o conjunto de todas as palavras dos KMs de M.
M é chamado de **Query Match** para Q se, e somente se, C é *cobertura total e minimal*

das palavras de Q. Isto é, $C{=}Q$ e $C \backslash \{K_{i,j}^{X} | 1 \leq j \leq m_i \wedge X \in \{S, V\}\} \neq Q$, $1{\leq}i{\leq}n$.

Como em geral existem vários QMs, desenvolvemos no *SEREIA* um algoritmo de ranqueamento baseado no algoritmo *CNRank* [Oliveira et al. 2015]. Este algoritmo utiliza modelos de linguagem com o objetivo de colocar nas primeiras posições do ranque os QMs com maior probabilidade de representar corretamente a intenção do usuário ao formular a consulta. Para isso, é calculado um escore de relevância para cada uma das QMs, que por sua vez, é baseado em um escore calculado para cada um de seus KMs. No caso de SKMs, a relevância é baseada em funções de similaridade entre palavras apresentadas na Seção 4.1. No caso de VKMs, a relevância é calculada com base na similaridade do cosseno, utilizando TF-IDF. Para computar o IDF, são calculadas as frequências máximas de cada atributo, além de sua norma.

4.3. Geração de Candidate Join Networks

Como dissemos, os QMs determinam os conjuntos de documentos de cada coleção que podem ser associados às palavras da consulta original. Quando corretamente vinculados, estes documentos satisfazem a consulta de forma total e minimal. Um exemplo de como os KMs de um QM podem ser associados neste sentido é apresentado no grafo J_1 abaixo. Ele é formado pelos KMs do Query Match C_1 da Seção 4.2 como nodos e suas arestas correspondem às associações entre coleções representadas no fragmento de banco de dados ilustrado na Figura 1. Especificamente, os documentos do KM $\texttt{REVIEW}^{S}[self^{\{reviewed\}}]$ são vinculados aos documentos do KM $\texttt{USER}^{V}[name^{\{michelle\}}]$ e também aos documentos do KM $\texttt{BUSINESS}^{V}[state^{\{Massachusetts\}}, categories^{\{italian,restaurants\}}]$.

$$J_1 = \texttt{USER}^{V}[name^{\{michelle\}}] \leftarrow \texttt{REVIEW}^{S}[self^{\{reviewed\}}]$$
$$\downarrow$$
$$\texttt{BUSINESS}^{V}[state^{\{Massachusetts\}}, categories^{\{italian,restaurants\}}]$$

Estes grafos, que na realidade são árvores, são chamados de Candidate Join Networks (CJNs) e correspondem a árvores de junções entre coleções de documentos. A partir das CJNs podem ser geradas consultas estruturadas que, quando executadas por um *document store* devem retornar o resultado corresponde a consulta por palavras-chave inicial. Como outro exemplo, a CJN J_2 abaixo pode ser derivada a partir da QM C_2 da Seção 4.2. Note-se que para cada QM gerada, podem haver uma ou mais CJNs derivadas.

$$J_2 = \texttt{USER}^{V}[name^{\{michelle\}}] \leftarrow \texttt{REVIEW}^{S}[self^{\{reviewed\}}]$$
$$\downarrow$$
$$\texttt{BUSINESS}^{S}[self^{\{restaurants\}}]^{V}[state^{\{Massachusetts\}}, name^{\{italian\}}]$$

Para definir Candidate Join Networks, precisamos de alguns definições iniciais. Um ***grafo de junções de Keyword Matches*** (GJKM) é um grafo da forma $J{=}\langle \mathcal{V}, E \rangle$, onde os nodos em $\mathcal{V}$ são Keyword Matches sobre coleções em um GD, e pode existir uma aresta $\langle KM_a, KM_b \rangle$ em E se existe uma ligação entre pelo menos um documento da coleção D_a e da coleção D_b, onde KM_x se refere à coleção D_x.

Dessa forma, uma ***Candidate Join Network*** para uma consulta Q é um GJKM $C{=}\langle \mathcal{V}, E \rangle$, que satisfaz algumas condições especiais. A principal delas é que os nodos em $\mathcal{V}$ contêm os KMs de um Query Match da consulta Q. Outras condições são exigidas para tornar o grafo minimal e evitar a geração de informações supérfluas na respostas. Por simplicidade, e sem perda de generalidade, omitiremos neste artigo o detalhamento sobre essas condições. Podemos, no entanto, adiantar que elas generalizam as condições necessárias para a existência de CJN em BDs relacionais, as quais são estudadas em

detalhes na literatura [Hristidis and Papakonstantinou 2002, Oliveira et al. 2020]. Essas condições são satisfeitas pelas CJN $J1$ e $J2$ apresentadas acima.

Ao final do processo, é necessário escolher uma das CJNs como sendo a mais adequada considerando a consulta inicial. Em nosso trabalho, tomamos as N QMs[6] com melhor posição no ranque de QM e geramos as CJNs correspondentes. Em seguida, ranqueamos todas as CJNs geradas, de forma que as CJNs mais concisas, ou seja, que envolvem menos coleções de documentos são melhor ranqueadas [Hristidis and Papakonstantinou 2002]. Deste ranque, selecionamos a primeira para gerar a consulta estruturada.

4.4. Geração de Consultas Estruturadas

No final do processo, uma das CJNs é transformada em uma consulta para ser executada sobre o GD. Nesse trabalho, como já foi dito, usamos o MongoDB, cuja linguagem de consulta segue um paradigma navegacional, e não declarativo como em SQL. Para dar suporte a junções, a linguagem adota o conceito de *aggregation*[7], que consiste em construir a resposta adicionando elementos dos documentos de uma coleção tomada como base, ou modificando elementos já existentes. Essa agregação pode conter múltiplos passos ou *stages*, cada uma realizando uma operação sobre a coleção formada no estágio anterior.

Em linhas gerais, nosso algoritmo de geração escolhe a coleção que corresponde a um dos VKMs das folhas da CJN e usa essa coleção como base. Para isso é usado um stage do tipo `$match`, que recupera apenas os documentos que satisfazem uma condição fornecida. Caso não ocorram VKMs nas folhas, é usado um stage do tipo `$project`. Em seguida, adicionam-se todas as outras VKMs e SKMs da CJN através de stages do tipo `$lookup`, que intuitivamente corresponde a uma junção parcial a esquerda da álgebra relacional. Depois, incluímos na consulta stages `$filter` para implementar as seleções referentes aos VKMs. Um efeito colateral disso é que os dados da coleções agregadas com `$lookup` e `$filter` ficam redundantes no resultado. Para evitar isso, incluímos stages `$project` pra remover os dados incluídos com `$lookup`. Um exemplo do resultado deste algoritmo é apresentado na Figura 3, que ilustra a consulta gerada para a CNJ J_1.

5. Resultados Experimentais

Nesta seção, apresentamos um experimento realizado com o *SEREIA* sobre o banco de dados Yelp![8], disponibilizado no formato JSON e que apresenta 6 coleções de documentos ricos em dados gerados por usuários. Para cada documento dentro de uma coleção há atributos que refletem as características da coleção e também atributos que referenciam documentos de outras coleções. A Tabela 1 apresenta alguns detalhes deste banco de dados. Para os experimentos, utilizamos um conjunto de 29 consultas adaptadas de um conjunto de consultas em linguagem natural a serem realizadas sobre um outro banco de dados contendo dados do Yelp!, mas que utiliza o modelo relacional [Yaghmazadeh et al. 2017]. Para avaliar o *SEREIA*, criamos conjuntos de referência de *Query Matches* e *Candidate Join Networks* que deveriam ser geradas pelo sistema para cada das 29 consultas. Os experimentos foram realizados em uma máquina com Ubuntu 20.04 64-bit, Intel Core

[6]Em nossos experimentos, usamos $N = 4$.

[7]Não confundir com o operador de agregação da álgebra relacional.

[8]`https://www.yelp.com/dataset`

i9-10850K 4.8GHz, 32GB RAM, 150GB SSD. Como document store, adotamos o MongoDB. As implementações foram feitas em Python 3.9.

Tabela 1. Estatísticas do dataset Yelp

Coleção	Número de documentos	Tamanho (GB)	Número de atributos	Número de atributos de referência	Número de referências existentes
USER	2.189.457	3,70	22	0	0
BUSINESS	160.585	0,12	21	0	0
REVIEW	8.635.403	6,90	9	2	17.270.806
TIP	1.162.119	0,23	5	2	2.324.238
CHECKIN	138.876	0,39	2	1	138.876
PHOTO	200.000	6,1	4	1	200.000
Média	2.083.073	2,9	10,5	1	3.322.320
Total	12.486.440	17,44	63	6	19.933.920

5.1. Resultados Gerais

A Tabela 2 ilustra estatísticas sobre as KMs, QMs e CJNs geradas. A quantidade de QMs geradas por consulta concentra-se no intervalo de 0 a 100. Em um caso específico, na consulta 8 ("*5 star italian restaurants*") há uma ocorrência maior de VKMs para os termos "*star*"e "*restaurants*", resultando em maior quantidade de combinações geradas entre termos. Por exemplo, o termo "*star*" pode se referir tanto à nota geral de um restaurante quanto a uma nota de REVIEW fornecida por um usuário. Da mesma forma, o termo "*restaurants*" pode se referir tanto ao nome de um restaurante quanto à uma categoria na coleção BUSINESS. Em relação a CJNs, como as KMs de uma QM pode ser conectadas de mais de uma forma, podem haver várias CJNs para uma mesma QM. Apesar disso, em alguns casos o número de CJNs é menor que o de QMs. Isso se deve ao fato de que algumas CJNs são podadas pelas restrições referidas na Seção 4.3.

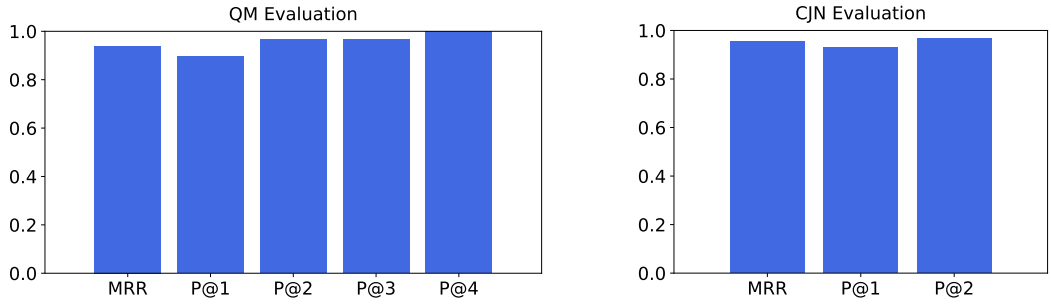
Tabela 2. Resultados Gerais

Consulta	VKMs	SKMs	QMs	CJNs	Consulta	VKMs	SKMs	QMs	CJNs
1	9	3	17	11	**16**	34	0	182	297
2	3	2	3	3	**17**	10	0	9	16
3	21	0	55	102	**18**	13	0	18	32
4	37	0	177	291	**19**	11	1	34	50
5	2	1	2	3	**20**	11	0	10	18
6	2	1	2	5	**21**	19	3	177	49
7	13	1	44	15	**22**	3	1	3	5
8	31	2	497	628	**23**	25	1	213	259
9	7	3	12	19	**24**	24	1	154	247
10	16	0	27	48	**25**	11	2	12	12
11	15	0	21	37	**26**	14	1	33	47
12	16	0	29	53	**27**	7	1	14	17
13	32	0	169	289	**28**	19	1	34	61
14	7	1	10	18	**29**	11	3	23	8
15	20	1	66	41	**Média**	13	1	27	92

5.2. Avaliação do Ranqueamento de Query Matches

Como já mencionado, e como visto na Tabela 2, uma consulta pode resultar em diversas QMs. Nesta seção verificamos a eficácia do algoritmo de ranqueamento de QM em selecionar as QMs mais adequadas. O primeiro ponto a destacar é que o algoritmo de geração de QM conseguiu encontrar as QMs corretas para todas as consultas. Na Figura 5a, apresentamos os resultados das métricas MRR e P@K. A métrica MRR é usada para computar a média das distâncias entre os itens relevantes do ranque e a primeira posição do ranque,

enquanto a métrica P@K mostra o percentual de itens relevantes até a posição K do ranque. Como pode ser visto, todas as QMs corretas são encontradas até a posição 4 do ranque, com P@4 com valor 1.0. No que se refere à métrica MRR, o *SEREIA* alcançou o valor de 0.94 para o dataset Yelp!. Esse é outro indicativo de que as respostas relevantes são identificadas majoritariamente nas primeiras posições do ranque de QMs.



(a) MRR e P@K de QMs **(b) MRR e P@K de CJNs**

Figura 5. Métricas de avaliação no ranqueamento de QMs e CJNs

5.3. Avaliação das Candidate Join Networks

Como descrito na Seção 4.3, as CJNs são geradas e ranqueadas a partir das N melhores QM. Neste experimento, usamos $N = 4$. Na Figura 5b, é possível ver que já na segunda posição, todas as CJNs corretas são encontradas, um ganho de 2 posições em relação ao P@K do ranque de QMs. Já a métrica MRR atingiu o valor de 0.98. Para efeito de exemplo, o não atingimento do valor máximo no MRR ocorre devido à consulta 1, "*business bricola 5 stars*", cuja CJN correta é a segunda nesse ranque.

5.4. Avaliação de Desempenho

Nesta seção, avaliamos o tempo gasto em todas as etapas, desde a geração de KMs até a obtenção de CJNs. Em média, a duração das etapas de geração de VKMs e SKMs durou 0.01 e 0.16 segundos, respectivamente. A duração da etapa de geração de QMs durou 0.16 segundos em média, enquanto a etapa de geração de CJNs durou, em média, 0.001 segundos. Lembramos que, no caso da geração de CJNs, selecionamos apenas um subconjunto de todas as QMs geradas, isto é, apenas as QMs com maiores possibilidades de satisfazerem a consulta fornecida. Por esse motivo, a etapa de geração de CJNs apresenta um tempo baixo. Dentre todas as consultas, apenas 2 consultas apresentaram tempo de geração maior a 1 segundo considerando todas as etapas. Nos dois casos, a maior quantidade de tempo gasto foi na geração de QMs, devido à maior quantidade de combinações possíveis a partir dos VKMs dessas duas consultas. Dentre essas consultas, o maior tempo de geração foi de 1,61 segundos.

6. Conclusões e Trabalhos Futuros

Apresentamos o *SEREIA*, um sistema que permite buscas por palavras-chave sobre *document stores*. O sistema se baseia no conceito de Candidate Join Networks, permitindo gerar diferentes interpretações de uma consulta de entrada e ranqueá-las para escolher a mais adequada. A avaliação experimental que realizamos sobre um banco de dados contendo várias coleções de documentos mostra que o *SEREIA* consegue gerar QMs e CJNs

corretas, além de ranqueá-las nas maiores posições do ranque e recuperar as CJNs corretas na primeira posição em 96% dos casos. Além disso, o sistema é capaz de gerar uma consulta a partir de uma CJN fornecida.

Este artigo se limitou a apresentar alguns resultados preliminares do *SEREIA*. Como trabalhos futuros, pretendemos estender as ideias e técnicas aqui propostas para realizar também consultas em linguagem natural sobre *document stores*. Além disso, vamos explorar o uso destas técnicas em sistemas de BDs multimodelos, ou *polystores*.

Referências

Baeza-Yates and Ribeiro-Neto (2011). *Modern Information Retrieval, Second Edition*. Pearson Education.

Bhalotia et al. (2002). Keyword searching and browsing in databases using banks. In *Proc. of 18th Intl. Conf. on Data Engineering*, pages 431–440.

Chasseur et al. (2013). Enabling json document stores in relational systems. In *Proc. of the 16th Intl. Workshop on the Web and Databases*, volume 13, pages 1–6.

DiScala and Abadi (2016). Automatic generation of normalized relational schemas from nested key-value data. In *Proc. of the 2016 Intl. Conf. on Management of Data*.

Hristidis and Papakonstantinou (2002). Discover: Keyword search in relational databases. In *VLDB'02: Proc. of the 28th Intl. Conf. on Very Large Databases*, pages 670–681.

Li and Jagadish (2014). Nalir: an interactive natural language interface for querying relational databases. In *Proc. of the 2014 ACM SIGMOD Intl. Conf. on Management of Data*, pages 709–712.

Liu et al. (2014). Json data management: supporting schema-less development in rdbms. In *Proc. of the 2014 ACM SIGMOD Intl. Conf. on Management of Data*.

Liu et al. (2016). Closing the functional and performance gap between sql and nosql. In *Proc. of the 2016 Intl. Conf. on Management of Data*, pages 227–238.

Oliveira et al. (2015). Ranking candidate networks of relations to improve keyword search over relational databases. In *2015 IEEE 31st Intl. Conf, on Data Engineering*, pages 399–410.

Oliveira et al. (2018). Match-based candidate network generation for keyword queries over relational databases. In *2018 IEEE 34th Intl. Conf. on Data Engineering (ICDE)*, pages 1344–1347.

Oliveira et al. (2020). Efficient match-based candidate network generation for keyword queries over relational databases. *IEEE Trans. on Knowledge and Data Engineering*.

Sadalage and Fowler (2012). Nosql distilled: A brief guide to the emerging world of polyglot persistence (2012). 13:978–0321826626.

Tahara et al. (2014). Sinew: a sql system for multi-structured data. In *Proc. of the 2014 ACM SIGMOD Intl. Conf. on Management of Data*, pages 815–826.

Yaghmazadeh et al. (2017). Sqlizer: Query synthesis from natural language. *Proc. ACM Program. Lang.*, (OOPSLA):1–26.