Universidade Federal do Amazonas

Instituto de Computação - ICOMP

Programa de Pós-Graduação em Informática

Lathe-DB - Integrando um Mecanismo de Busca por Palavras-Chave em SGBDs Relacionais

Gilberto Eduardo Santos

Manaus - AM

Abril de 2018

Universidade Federal do Amazonas
Instituto de Computação - ICOMP
Programa de Pós-Graduação em Informática

Lathe-DB - Integrando um Mecanismo de Busca por Palavras-Chave em SGBDs Relacionais

Gilberto Eduardo Santos

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Computação - ICOMP, da Universidade Federal do Amazonas, como parte dos requisitos necessários à obtenção do titulo de Mestre em Informática.

Orientador: Altigran Soares da Silva

Manaus - AM
Abril de 2018

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Santos, Gilberto Eduardo
S237l Lathe-DB - Integrando um Mecanismo de Busca por Palavras-Chave em SGBDs Relacionais / Gilberto Eduardo Santos. 2018
89 f.: il. color; 31 cm.

Orientador: Altigran Soares da Silva
Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. Busca por palavras-chave. 2. banco de dados. 3. sql. 4. sgbd. 5. relacional. I. Silva, Altigran Soares da II. Universidade Federal do Amazonas III. Título

FOLHA DE APROVAÇÃO

"Lathe-DB - Integrando um Mecanismo de Busca por Palavras-Chave em SGBDs Relacionais"

GILBERTO EDUARDO SANTOS

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Altigran Soares da Silva - PRESIDENTE

Profa. Rosiane de Freitas Rodrigues - MEMBRO INTERNO

Prof. João Marcos Bastos Cavalcanti - MEMBRO EXTERNO

Manaus, 03 de Abril de 2018

A Deus e em especial aos meus pais ***Claudenir Santos*** *e* ***Maria Aparecida Santos*** *que sempre me apoiaram em todas as etapas acadêmicas e incentivaram-me a nunca desistir*

Agradecimentos

Agradeço, primeiramente, a Deus por me proporcionar saúde e os meios necessários para realizar esse trabalho.

Ao meu orientador Prof. Dr. Altigran Soares da Silva por sua paciência, ensinamentos e conhecimentos repassados que possibilitaram a realização deste trabalho de pesquisa.

Aos colegas de Mestrado, Diego Barros e Leonardo Nascimento que muito me auxiliaram nas horas de dificuldades e ao Dr. Péricles Oliveira, pelas dúvidas esclarecidas.

À minha namorada Lígia Batista Galvão, que me apoiou durante a realização desse trabalho e também durante os problemas de saúde enfrentados ao longo desse período. Ao meu filho Phelipe Eduardo Santos pelo apoio e incentivo.

À Universidade Federal do Amazonas - UFAM, pela utilização de seus laboratórios e equipamentos durante o desenvolvimento desse projeto.

À FPF Tech por ter flexibilizado meu horário de trabalho e me liberando durante o expediente para compromissos na Universidade, em especial ao amigo Luis Braga, que me incentivou e apoiou na realização desse trabalho.

Enfim, a todos aqueles que contribuíram direta ou indiretamente com esse trabalho.

'Não importa quanto a vida possa ser ruim, sempre existe algo que você pode fazer, e triunfar.
Enquanto há vida, há esperança.'
(Stephen Hawking)

Resumo

Durante muitos anos, vários pesquisadores propuseram métodos para capacitar os usuários leigos, sem qualquer conhecimento em linguagens de consulta ou sobre os bancos de dados, a realizar consultas em Sistemas Gerenciadores de Bancos de Dados - SGBD usando palavras-chave. Sistemas que processam consultas baseadas em palavras-chaves em SGBDs, comumente chamados de *Relational Keyword-Search* - R-KwS, enfrentam a tarefa de determinar automaticamente, a partir de algumas palavras-chave, quais informações devem ser recuperadas do banco de dados, e como essas informações podem ser combinadas para fornecer uma resposta relevante, e de qualidade, para o usuário. Porém, uma característica comum, na grande maioria dos atuais sistemas R-KwS, é que a implementação desses sistemas é feita geralmente fora do ambiente do SGBD. Isso exige, para o seu funcionamento, uma infraestrutura de software separada, com instalação complexa e posterior dificuldade para realizar manutenções. O objetivo principal desta dissertação é integrar um mecanismo de busca por palavras-chave em um SGBD, implementando o método Lathe, proposto no grupo de Banco de Dados e Recuperação da Informação do ICOMP, no ambiente do SGBD PostgreSQL. Experimentos realizados, e cujo os resultados apresentamos nessa dissertação, mostram que a implementação do Lathe-DB, como um mecanismo de busca por palavras-chaves dentro do ambiente do SGBD, geram resultados bem próximos aos do trabalho original, e que é uma forma bem prática, de baixo custo e que agrega valor, para o uso da busca por palavras-chave em SGBDs.

Palavras-chave: Busca por palavras-chave, Bancos de Dados, SQL, SGBD.

Abstract

For many years, several researchers have proposed methods to empower the naive users, without any knowledge about query languages or database details, to perform queries in RDBMS using keywords.ystems that process queries based on keywords in RDBMS, commonly called Relational Keyword-Search - R-KwS , face the task of automatically determine, from some keywords, what information should be retrieved from the database, and how this information can be combined to provide a relevant and quality answer to the user. However a common feature, in the vast majority of current R-KwS systems, is that the implementation of these systems is usually done outside the RDBMS environment. This requires, for its operation, a separate software infrastructure, with complex installation and subsequent hard to perform maintenance. The main objective of this dissertation is integrate a search engine for keyword in a RDBMS, implementing the Lathe method, proposed in the database and information retrieval group of ICOMP, inside PostgreSQL RDBMS environment. Experiments show that the implementation of Lathe-DB, as a search engine inside the RDBMS environment, generate results very close to those of the original work, and that is a very practical, low cost, and adds value, for use the keyword search in RDBMS.

Keywords: Keyword Search, Databases, SQL, RDBMS.

Sumário

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
ANSI	American National Standards Institute
CN	*Candidate Network*
FK	*Foreign Key* - chave estrangeira
JNT	*Joining Network of Tuples*
MJNT	*Minimal Joining Network of Tuples*
PK	*Primary Key* - chave-primária
R-KwS	*Relational Keyword-Search*
RIR	Restrições de Integridade Referencial
SGBD	Sistema Gerenciador de Banco de Dados
SQL	*Structured Query Language*

1 Introdução

A maioria das empresas, ou qualquer instituição pública ou privada, necessita armazenar seus dados de forma estruturada e organizada para que seja possível, além de recuperar rapidamente esses dados, gerar informação de valor para tomada de decisão. Diante dessa necessidade, o uso de Sistemas Gerenciadores de Bancos de Dados - SGBDs foi amplamente adotado pelas mesmas, pois permite que os dados sejam facilmente armazenados e recuperados. Porém, recuperar essas informações requer usuários com conhecimentos especializados nesses sistemas.

Durante muitos anos, vários pesquisadores propuseram métodos para capacitar os usuários leigos, sem qualquer conhecimento em linguagens de consulta ou sobre os banco de dados, a realizar consultas em SGBDs usando palavras-chave [Aditya et al. 2002, Agrawal, Chaudhuri e Das 2002, Hristidis e Papakonstantinou 2002, Hristidis, Gravano e Papakonstantinou 2003, Agrawal et al. 2003, Mesquita et al. 2007, Fan, Li e Zhou 2011, Li et al. 2010, Oliveira e Silva 2015, Oliveira e Silva 2017]. As motivações para isto são muitas e podemos destacar as seguintes:

- a busca por palavras-chave é o principal método de pesquisa por documentos na Internet e é familiar ao usuário comum. (ex. Google, Yahoo, Bing, Lycos, Excite, etc.);

- atualmente, recuperar informações em SGBDs exige conhecimento especializado na linguagem SQL, no esquema do banco de dados e nas particularidades de cada SGBD;

- a solução mais comum para disponibilizar as informações contidas em SGBDs é o desenvolvimento de aplicações específicas, onde campos de consulta são fornecidos para que os usuários realizem consultas pré-definidas para realizar as consultas. Essa solução, além de não ser flexível, cria uma enorme dependência com desenvolvedores de aplicativos;

Esses sistemas enfrentam a tarefa de determinar automaticamente, a partir de algumas palavras-chave, quais informações devem ser recuperadas do banco de dados, e como essas informações podem ser combinadas para fornecer uma resposta relevante, e de qualidade, para o usuário. De fato, esta é uma tarefa desafiadora uma vez que a informação procurada, frequentemente está dispersa por várias tuplas e tabelas, dependendo do projeto do esquema do banco de dados que se deseja pesquisar.

Sistemas dessa natureza são comumente chamados de Sistemas de Pesquisa por Palavras-Chave Relacionais - *Relational Keyword Search Systems* - R-KwS e normalmente se encaixam em uma de duas categorias distintas:

- baseados em Grafos do Esquema - *Schema Graphs* - Nessa abordagem, o banco de dados é modelado como um grafo, onde os vértices são as tabelas e cada aresta representa uma Restrição de Integridade Referencial -RIR;
- baseados em Grafos de Dados - *Data Graphs* - Nessa abordagem, o banco de dados é modelado como um grafo, onde cada vértice representa uma tupla e as arestas representam as instâncias das RIR;

Umas das principais diferenças entre os sistemas baseados em *Schema Graphs* e os baseados em *Data Graphs* é que os sistemas do primeiro tipo recebem como entrada uma consulta com palavras-chave e produzem uma consulta SQL que será processada por um SGBD, de forma que a resposta a esta consulta satisfaz a consulta original. Na literatura, essas consultas são chamadas de Redes de Tabelas ou *Network of Relations* [Hristidis e Papakonstantinou 2002]. Assim, sistemas desse tipo, tiram proveito de todos os recursos já disponíveis em um SGBD. Ao contrário, os sistemas do segundo tipo são eles próprios responsáveis pelo processamento das consultas feitas com palavras-chave. Por essa razão, nesta dissertação vamos apresentar uma contribuição que aborda sistemas R-KwS da primeira categoria, a baseada em *Schema Graphs*.

Uma característica comum na grande maioria dos atuais sistemas R-KwS, em qualquer uma das categorias mencionadas, é que sua implementação é feita geralmente fora do ambiente do SGBD. Isso exige, para o seu funcionamento, uma infraestrutura de software separada, com instalação complexa e posterior dificuldade para realizar manutenções. Que seja do nosso conhecimento, identificamos apenas um trabalho previamente proposto na literatura não tem essa característica, o CSTree [Li et al. 2010], que foi implementado diretamente no ambiente do SGBD.

O objetivo principal desta dissertação é integrar um mecanismo de busca por palavras-chave em um SGBD, implementando o método proposto no sistema Lathe [Oliveira e Silva 2015, Oliveira e Silva 2017], no ambiente do SGBD PostgreSQL.

O sistema Lathe foi escolhido porque os resultados apresentados pelos autores indicam melhora no desempenho das buscas e fornecem respostas de qualidade superior em comparação a outros métodos. Acreditamos que esses resultados se devem a nova abordagem para a geração de Redes de Tabelas com base no conceito de *Steiner Trees* [Dreyfus e Wagner 1971]. Embora o uso do conceito de *Steiner Trees* seja comum em sistemas R-KwS baseados em *Data Graphs*, essa abordagem não foi

considerada antes para o problema da geração de Redes de Tabelas que são a base para sistemas da categoria *Schema Graphs*. Além disso, o método foi desenvolvido em nosso grupo de pesquisa, o que serve de motivação adicional para mostrar sua aplicabilidade em um ambiente de SGBD.

Apesar do sistema CSTree [Li et al. 2010] ter uma proposta semelhante a do nosso trabalho, este difere essencialmente na categoria implementada, uma vez que é baseado no conceito de *Data Graphs* e sua principal contribuição foca em apresentar o um novo conceito chamado Árvores Compactas de Steiner - *Compact Steiner Tree*. Deste modo a semelhança é somente na implementação do trabalho no ambiente do SGBD.

1.1 Motivações

Nossas motivações para realizar esse trabalho são:

- aproveitar os recursos e a robustez típicas dos SGBDs, tais como: Índices, *triggers*, *views*, funções, transações, propriedades ACID e etc.;
- diminuir a complexidade da infraestrutura de software necessária para a utilização de um R-KwS;
- ajudar o desenvolvedor de sistemas a prover a busca por palavras-chave nos seus aplicativos com mais facilidade;
- promover uma rápida adoção da busca por palavras-chave em bancos de dados, mesmo os já em produção;

Para isso, enfrentamos os seguintes desafios:

- recursos internos do SGBD diferem do ambiente externo ao mesmo, o seja, não temos a mesma flexibilidade para execução de programas em um sistema operacional típico;
- utilização somente de uma linguagem do tipo PL/SQL, procurando seguir o máximo possível do padrão ANSI, com o intuito de tornar a implementação facilmente portável para outros SGBDs;
- limitação das tecnologias atuais dos SGBDs, as quais não foram projetadas para suportar busca por palavras-chave;

1.2 Lathe-DB

Neste trabalho foi desenvolvido um mecanismo de consultas por palavras-chave, chamado Lathe-DB, que, ao contrário da maioria dos R-KwS propostos na literatura, é embutido no próprio ambiente do SGBD alvo. Baseado no método Lathe [Oliveira e Silva 2015, Oliveira e Silva 2017], o nosso mecanismo foi totalmente desenvolvido no ambiente do SGBD PostgreSQL 9.5 padrão, usando somente PL/pgSQL, sem o uso de qualquer módulo externo e não alterando a estrutura original banco de dados que se deseja consultar. O uso do Lathe-DB requer somente a criação de algumas tabelas e funções PL/pgSQL em um esquema em separado do banco de dados original.

O Lathe-DB foi concebido para ser facilmente utilizado. Como exemplo, considere a consulta 'denzel washington gangster' e vamos executá-la no banco de dados do IMDb[1], que é um banco de dados sobre filmes bem conhecido da Internet.

Como podemos observar na Figura 1, o uso do mecanismo de busca no dia-a-dia é bem simples. Só é necessário utilizar uma consulta, que executa a função matCnGen ①, localizada no esquema latheDB, informando como parâmetro uma sequência de caracteres que contém o conjunto de palavras-chave que serão pesquisadas, no exemplo 'denzel washington gangster' ②. Durante a execução o Lathe-DB exibe algumas mensagens ③ que mostram os passos que estão sendo executados, afim de apresentar ao usuário a evolução do processo.

A partir dai, o Lathe-DB gera uma ou mais possíveis redes de tabelas, aqui chamadas de Redes Candidatas - *Candidates Network (CN)*, que são consultas SQL cuja semântica é a mesma da consulta original, ou seja, ela produz como resultado as tuplas que melhor satisfazem a consulta original com palavras-chave.

Para nossa consulta de exemplo, as duas CNs geradas são exibida na parte inferior da Figura 1 ④ e, como podemos observar, foi utilizado o operador *ilike* para buscar os termos da consulta dentro do conteúdo textual do atributos. Além disso, esse operador não faz distinção entre letras maiúsculas ou minúsculas. Existem outras formas de realizar essa operação, que são suportadas pelo mecanismo do Lathe-DB, que são descritas no Capítulo 4. As CNs geradas são classificadas, utilizando o algoritmo CNRank [Oliveira, Silva e Moura 2015], e as top-k CNs são retornadas para o usuário.

Formalmente, CNs são expressões em álgebra relacional que fazem a conexão de árvores de tabelas derivadas a partir do grafo que representa o esquema relacional do banco de dados alvo. Como podemos observar, a Equação 1.1 corresponde à primeira

[1] Nesse exemplo utilizamos um subconjunto do banco IMDb de 2009, que pode ser obtido em https://www.cs.virginia.edu/ jmc7tp/resources.php

```
imdb2009_subset=# SELECT * FROM latheDB.matCnGen('denzel washington gangster');
NOTICE:  Creating TupleSets ...  (1)
NOTICE:  5 tupleset(s) found.
NOTICE:  Generating matches ...  (2)
NOTICE:  1010 Matches created.  (3)
NOTICE:  Ranking CNs...
NOTICE:  Finding Steiner Nodes ...
                                candidate_network
-------------------------------------------------------------------------------
 SELECT name.name, title.title FROM title, cast_info, name WHERE
title.id =cast_info.movie_id and cast_info.person_id=name.id AND
name.name ilike'%denzel%' and name.name ilike'%washington%' AND
title.title ilike'%gangster%';
 SELECT * FROM cast_info,movie_info,name,title;
WHERE cast_info.movie_id=title.id AND cast_info.person_id=name.id AND
movie_info.movie_id=title.id AND  name.name ilike'%denzel%' AND movie_info.info
ilike'%gangster%' and movie_info.info ilike'%washington%';
(2 rows)
                                                    (4)

imdb2009_subset=#
```

Figura 1 – Execução do Lathe-DB.

CN da Figura 1 ④, na forma de expressão em álgebra relacional.

$$\sigma_{\substack{\text{TITLE}\\ \text{title}\supseteq\{gangster\}}} \bowtie_{\text{id=movie\_id}} \text{CAST\_INFO} \bowtie_{\text{person\_id=id}} \sigma_{\substack{\text{NAME}\\ \text{name}\supseteq\{denzel\ washington\}}} \tag{1.1}$$

Na Figura 2 apresentamos o resultado da execução da primeira CN no SGBD. Neste caso, uma única tupla é retornada, que semanticamente satisfaz a consulta original, trazendo como resposta o filme com o título *"American Gangster* , estrelado por *Denzel Washington*.

```
        name         |        title
---------------------+--------------------
 Washington, Denzel | American Gangster
(1 row)

imdb2009_subset=#
```

Figura 2 – Execução da *Candidate Network*.

Experimentos realizados mostram que a implementação do Lathe-DB, como um mecanismo de busca dentro do ambiente do SGBD geram resultados bem próximos aos do trabalho original, proposto no sistema Lathe [Oliveira e Silva 2015, Oliveira e Silva 2017]. Também mostra que é uma forma mais prática, de baixo custo de manutenção, para o uso da busca por palavras-chave em SGBDs, uma vez que aproveita a infraestrutura já existente.

1.3 Organização do Trabalho

A presente dissertação está estruturada como segue: após este capítulo introdutório, este trabalho consiste em outros quatro capítulos. O Capítulo 2 apresenta a fundamentação teórica e os trabalhos relacionados, explicando em detalhes as duas principais vertentes adotadas pelos pesquisadores sobre a busca por palavras-chave em bancos de dados relacionais, e apresenta alguns dos principais trabalhos relacionados. Para manter esta dissertação auto-contida, dedicamos o Capítulo 3 para o Lathe, trabalho em que é baseado a solução proposta neste trabalho, o Capítulo 4 descreve em detalhes o Lathe-DB e seu funcionamento. O Capítulo 5 apresenta uma análise experimental comparativa com o Lathe e o Lathe-DB. E finalmente, o Capítulo 6 expõe as conclusões e trabalhos futuros.

2 Fundamentação Teórica e Trabalhos Relacionados

Neste capítulo, apresentamos a fundamentação teórica necessária ao entendimento do trabalho e descrevemos de forma resumida os principais trabalhos relacionados na literatura. Iniciamos pela apresentação dos conceitos e terminologias usadas em sistemas *Relational Keyword Search* - R-KwS .

2.1 Conceitos e Terminologia

Os conceitos e terminologia que utilizamos neste trabalho foram apresentados inicialmente no sistema DISCOVER [Hristidis e Papakonstantinou 2002], e foram utilizados pela maioria dos trabalhos baseados neste. Em particular, utilizamos as definições de *Joining Network of Tuples, Minimal Total Joining Network of Tuples, TupleSets, Keyword Query* e *Candidate Networks*. Por uma questão de conveniência, algumas definições a seguir serão apresentas com ligeiras modificações com relação ao sistema DISCOVER.

Assim como no sistema DISCOVER, considere um grafo de esquema G que representa um esquema relacional, onde os vértices correspondem as tabelas e as arestas correspondem as Restrições de Integridade Referencial (RIR) entre as tabelas, como mostra a Figura 3. Para as definições a seguir, as direções das RIR não são importantes, então nós consideramos uma versão não direcionada G_u do grafo G.

Definição 1: Uma Junção de Redes de Tuplas - ***Joining Network of Tuples (JNT)*** j é uma árvore de tuplas onde cada par de tuplas adjacentes $t_i, t_j \in j$, onde t_i and t_j são tuplas das tabelas R_i e R_j, respectivamente, existe uma aresta $\langle R_i, R_j \rangle$ em G_u e $(t_i \bowtie t_j) \in (R_i \bowtie R_j)$.

Definição 2: Dado um conjunto $Q = \{k_1, \ldots, k_n\}$ de palavras-chave, uma JNT j é uma Mínima e Total JNT - ***Minimal Total JNT (MTJNT)*** para Q se a mesma é simultaneamente **total**, isto é, cada palavra-chave w_i esta contida em pelo menos uma tupla de j, e **mínima**, isto é, qualquer JNT j' que resulta da remoção de qualquer tupla de j não é total.

Definição 3: Uma Consulta por Palavra-Chave - ***Keyword Query*** é uma lista Q de palavras-chave cujo resultado é: (i) o conjunto T de tuplas de uma tabela R, de tal forma que cada uma de suas tuplas contêm todas as palavras-chave de Q, ou (ii) um conjunto M de todas as possíveis MTJNT para as palavras-chave em Q sobre um

conjunto de tabelas $\{R_1, \ldots, R_m\}$.

Definição 4: Seja Q uma *Keyword Query* e seja K um subconjunto de Q. Além disso, seja R_i uma tabela. Um Conjunto de Tuplas - ***Tuple-Set*** de R_i sobre K é dado por:

$$R_i^K = \{t | t \in R_i \wedge \forall k \in K, k \in W(t) \wedge \forall \ell \in Q = K, \ell \notin W(t)\} ,$$

onde $W(t)$ fornece o conjunto de termos em t. Se $K = \emptyset$, o *tuple-set* é chamado de *Free Tuple-Set* e é denotado por $R_i^{\{\}}$. De acordo com a Definição 4, R_i^K contêm as tuplas de R_i que contêm todos os termos de K e nenhum outro termo de Q.

Definição 5: Uma Junção de Redes de Conjuntos de Tuplas - ***Joining Network of Tuple-Sets*** J é uma árvore de *tuple-sets* onde cada par de *tuple-sets* adjacentes R_i^K, R_j^M em J é uma aresta $\langle R_i, R_j \rangle$ no grafo G_u.

Definição 6: Dado um conjunto de *Keyword Query* $Q = \{k_1, \ldots, k_n\}$, uma Rede Candidata - ***Candidate Network (CN)*** C é uma árvore de *tuple-sets*, onde cada par adjacente de *tuple-sets* R_i^K, R_j^M em C é uma aresta $\langle R_i, R_j \rangle$ em G_u. Além disso, C deve ser *total*, isto é, cada palavra-chave em Q deve esta contida em pelo menos um *tuple-set* de C, e *minimal*, isto é C não é *total* se qualquer *tuple-set* for removido.

O procedimento para a geração de CNs para uma *keyword query* fornecida pelo usuário consiste em, primeiro procurar todos os *tuple-sets* que contêm as palavras-chave da consulta e então percorrer o grafo do esquema do banco de dados tentando conectar esses *tuple-sets*. Cada CN é portanto, uma árvore, cujas as folhas são os *tuple-sets* e os nós internos são os *free tuple-sets* que os conectam. Intuitivamente, CNs são expressões em álgebra relacional que unem tabelas a partir do grafo que representa o esquema relacional. Em outras palavras cada CN é uma consulta SQL que, usando as Restrições de Integridade Referencial (RIR), conecta as tabelas do banco de dados e descreve como produzir respostas plausíveis para a consulta por palavras-chave fornecida pelo usuário. O melhor algoritmo conhecido para a geração de CN é o CNGen, proposto no sistema DISCOVER [Hristidis e Papakonstantinou 2002].

2.2 Sistemas R-KwS baseados em *Schema Graph*

Sistemas R-KwS podem ser classificados em duas categorias: os baseados em *Data Graphs* e os baseados em *Schema Graphs*. O foco desta dissertação são os sistemas dessa última categoria.

Em sistemas R-KwS baseados em *Schema Graphs*, o esquema do banco de dados é modelado como um grafo $G(V, E)$, onde V é o conjunto R de tabelas do banco de dados, e cada tabela R_i é representada como um vértice no grafo, e E é o conjunto de arestas do grafo, onde cara aresta representa uma Restrição de Integridade Referencial

(RIR), na forma $R_i \rightarrow R_j$.

Como exemplo, apresentamos na Figura 3 um esquema relacional[1], baseado em uma amostra do banco de dados IMDB[2] [Coffman e Weaver 2010], e sua representação em forma de grafo logo abaixo. Neste exemplo a tabela MOVIE armazena os nomes dos filmes, CHAR armazena os nomes dos personagens dos filmes, NAME os nomes dos pessoas que participaram dos filmes, ROLE contém a função exercida pelas pessoas nos filmes e a tabela CAST, que associa todas as demais tabelas formando o elenco e a produção dos filmes. Logo abaixo, na representação em forma de grafo, é fácil observar que cada tabela do banco dados tem um uma correspondência direta para cada vértice do grafo e as Restrições de Integridade Referencial (RIR), entre chaves primárias e estrangeiras, são representadas pelas arestas do grafo.

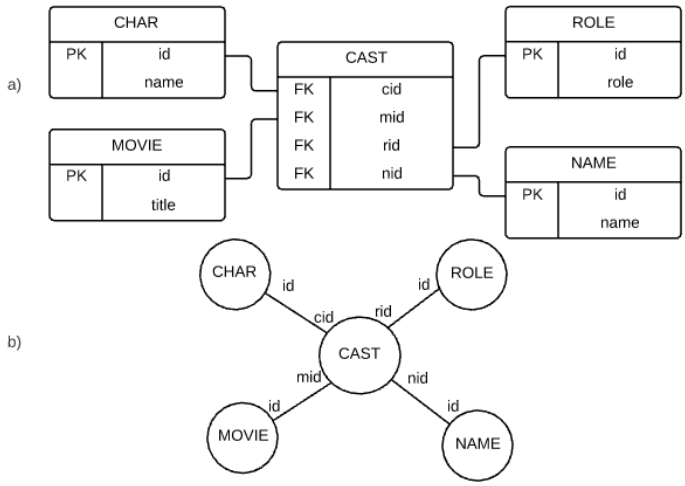


Figura 3 – a) Esquema relacional do banco de dados IMDB e b) sua representação em forma de grafo.

2.3 Arquitetura de sistemas R-KwS baseados em *Schema Graph*

Nos sistemas baseados em *Schema Graphs*, assumimos que os dados que desejamos consultar estão armazenados em um SGBD e que este é capaz de executar consultas SQL. A Figura 4 apresenta o funcionamento e arquitetura típica de um sistema R-KwS baseado em *Schema Graphs* que descrevemos a seguir.

Inicialmente, na Figura 4, uma *keyword query* não-estruturada é submetida pelo usuário, normalmente através de uma interface simples, com uma caixa de texto

[1] A notação utilizada para representar o esquema relacional do banco de dados é o mesmo utilizado nas ferramentas CASE (Computer-Aided Software Engineering), disponíveis no mercado, utilizadas nas atividades de modelagem de dados.

[2] Os nomes das tabelas e atributos foram alterados para melhor visualização.

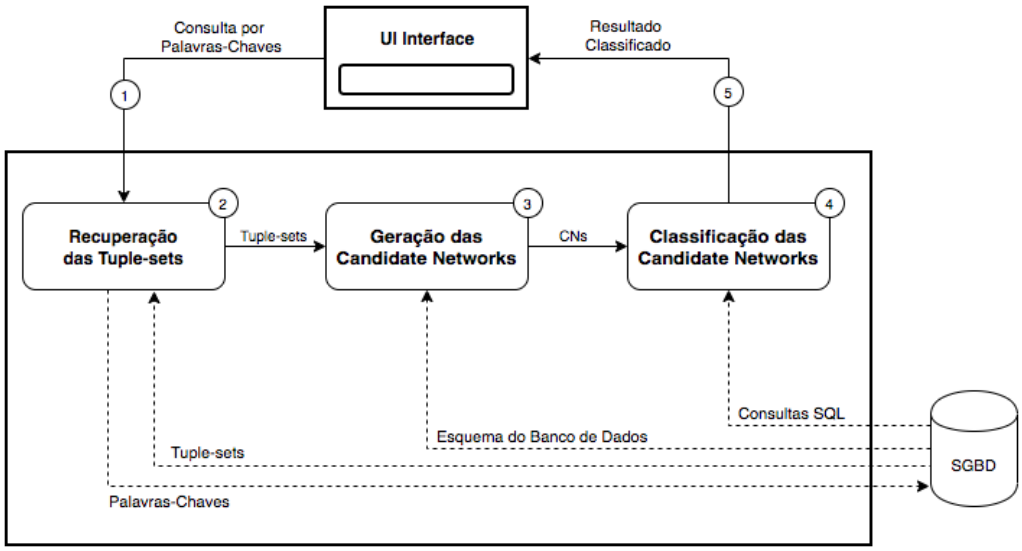


Figura 4 – Arquitetura típica de um R-KwS baseado em *Schema Graph*.

①. A *keyword query* é enviada pelo usuário que tem como objetivo buscar tuplas interconectadas que contêm as palavras-chave. É dito que uma tupla que contém uma palavra-chave se o conteúdo de um atributo contém a palavra-chave da consulta.

De posse das palavras-chave, o sistema procura por subconjuntos de tuplas que contêm as palavras-chave da consulta. Esses subconjuntos, chamados *Tuple-Sets*, são obtidos a partir do banco de dados ② percorrendo todas as suas tabelas. A seguir esses *tuple-sets* são usados para gerar as *Candidade Networks* (CNs) ③, essa abordagem é proposta nos sistemas DISCOVER [Hristidis e Papakonstantinou 2002] e DBXplorer [Agrawal, Chaudhuri e Das 2002], e mais tarde adotado por diversos outros sistemas como o Efficient [Hristidis, Gravano e Papakonstantinou 2003], SPARK [Luo et al. 2007], CD [Coffman e Weaver 2010], Min-cost [Ding et al. 2007], S-KwS [Markowetz, Yang e Papadias 2007] e KwS-F [Baid et al. 2010].

Além dos *tuple-sets* gerados no passo anterior, a geração das CNs também requer informação adicional sobre as restrições de integridade referencial, essa informação é obtida através do esquema do banco de dados. Uma vez que existem diferentes caminhos para unir as tabelas que estão armazenadas as tuplas que contém as palavras-chave, várias CNs diferentes podem ser potencialmente geradas. Na prática porém, somente algumas são realmente úteis para produzir respostas de qualidade para a consulta do usuário. O mais conhecido algoritmo para a geração de CNs é chamado *CNGen*, e foi proposto no sistema DISCOVER. No próximo passo ④ as CNs geradas são avaliadas a fim de obter respostas do banco de dados a consulta fornecida pelo usuário. Nesse contexto, as respostas são as *Joining Networks os Tuples* (JNTs) [Hristidis e Papakonstantinou 2002], isto é, árvores compostas por tuplas unidas por suas RIR

que contém as palavras chaves. Vários algoritmos diferentes tem sido propostos para avaliar CNs, como nos sistemas DISCOVER, DBXExplorer e no S-KwS. Em particular, no estado-da-arte, sistemas como o Efficient, SPARK e o CD, somente as *top-K* JNTs são retornadas, o que requer que essas sejam classificadas utilizando funções de Recuperação da Informação (RI). Finalmente as *top-K* respostas são então apresentadas para o usuário ⑤. A principal motivação para classificar as JNTs em sistemas como o Efficient, SPARK e o CD é evitar problemas de otimização consultas, como ocorre no sistema DISCOVER [Hristidis e Papakonstantinou 2002].

O problema da eficiência e escalabilidade na avaliação de CN é tratado de uma forma diferente no sistema KwS-F [Baid et al. 2010]. Essa abordagem consiste em dois passos. Primeiro, um limite para o tempo gasto na avaliação das CNs. Após esse limite ser alcançado, o sistema deve retornar (possivelmente parcialmente) as *top-K* JNTs resultantes. Em segundo lugar, se existirem CNs ainda a serem avaliadas, estas CNs são apresentados ao usuário por meio de formulários de consulta, de modo que o usuário pode selecionar quais CNs deseja avaliar e o sistema avalia a CN correspondente. No sistema KwS-F, os autores relatam uma série de descobertas experimentais sobre a eficácia e viabilidade da abordagem proposta mas, infelizmente, eles não informam sobre a qualidade dos resultados obtidos.

No sistema S-KwS [Markowetz, Yang e Papadias 2007], os autores apresentam a estratégia de ordenar os nós internos da CNs geradas pelo algoritmo CNGen visando detectar possíveis CNs duplicadas. Isto pode reduzir o número de CNs manipuladas pelos sistemas, mas ainda requer que todas as CNs, duplicadas ou não, sejam geradas.

No trabalho *"A Framework for evaluating databases keyword search strategies"* [Coffman e Weaver 2010], ot autores propõem um *framework* para avaliar sistemas R-KwS e reportaram o resultado da aplicação desse *framework* sobre três conjuntos de dados padronizados, chamados *Mondial*, *IMDb* e *Wikipedia* [Coffman e Weaver 2010], juntamente com suas respectivas consultas e cargas de trabalho. Os autores compararam nove sistemas R-KwS no estado-da-arte, e os avaliaram em muitos aspectos relacionados com performance e eficácia. Uma importante contribuição deste trabalho é que, em termos de eficácia, nenhum dos sistemas testados obteve o melhor desempenho em todos os conjuntos de dados para os conjuntos de consultas propostas. Até agora, este é o único estudo na literatura que fornece avaliações qualitativas dos sistemas R-KwS. Na nossa avaliação experimental usamos os conjuntos de dados e os conjuntos de consultas apresentados neste trabalho.

2.4 Sistemas R-KwS

A seguir apresentamos alguns trabalhos que serviram como fonte de informação e dedicamos um capítulo especial ao Lathe [Oliveira e Silva 2015, Oliveira e Silva 2017], que é a base da nossa implementação.

2.4.1 DISCOVER

O sistema DISCOVER [Hristidis e Papakonstantinou 2002] é um trabalho da categoria *Schema Graph* que implementa a abordagem de *Candidate Networks* (CNs), e produz, sem redundância todas as CNs relevantes para a consulta do usuário, isto é, todas as CNs que produzem respostas que semanticamente satisfação a pesquisa do usuário . Durante a avaliação das CNs, o DISCOVER possui um módulo gerador de plano de execução que utiliza resultados intermédios para minimizar o custo total da avaliação de todas as CNs. Porém o problema de selecionar o melhor conjunto de resultados intermédios é NP-completo, dessa forma o DISCOVER apresenta um algoritmo guloso que descobre o plano execução quase ideal para a consulta do usuário.

2.4.2 BANKS

O sistema BANKS [Aditya et al. 2002], é uma ferramenta da categoria *Data Graph* que, conforme já mencionado, modela um grafo a partir da instância dos dados, que são os vértices e as restrições de integridade referencial como arestas desse grafo, porém arestas adicionais podem ser criadas para representar outras tabelas de dependência entre as tuplas. O BANKS ainda atribui pesos nas arestas e vértices do grafo: nas arestas conforme a proximidade, relevância e tipo de ligação e nos vértices conforme o número de arestas que apontam para para cada vértice, o que determina sua relevância. Esses pesos não utilizados para classificar as respostas as pesquisas feitas pelos usuários.

2.4.3 DBXplorer

O sistema DBXplorer, [Agrawal, Chaudhuri e Das 2002], ferramenta da categoria *Schema Graph*, que também implementa a abordagem baseada em CNs, utiliza uma estrutura própria, chamada Tabela de Símbolos, para mapear os locais onde se encontram os termos da consulta no banco de dados. A criação da Tabela de Símbolos é feita em uma etapa de pré-processamento e utiliza uma estrutura *hash* para encontrar rapidamente os mapeamentos e então monta um grafo com o conjunto das tabelas relevantes aos termos da consulta.

2.4.4 Efficient

O sistema Efficient [Hristidis, Gravano e Papakonstantinou 2003] difere do DISCOVER, seu antecessor, pois utiliza técnicas de Recuperação da Informação (RI), que são incorporadas nos SGBDs modernos, para classificação dos conjuntos de tuplas que serão utilizados para a montagem das *Candidate Networks* (CNs), enquanto o DISCOVER [Hristidis e Papakonstantinou 2002] a classificação é feita apenas pela proximidade dos termos da pesquisa, ou seja, quanto mais próximos os termos estiverem no bancos de dados, mais relevantes os mesmos são para a consulta do usuário.

O Efficient também inova ao trabalhar com as semânticas AND e OR enquanto o DISCOVER trabalha apenas com a semântica AND, isso que dizer que o Efficient poder retornar resultados que não contenham todas as palavras-chave da consulta, porém, com bons resultados por utilizar com um alto grau de relevância devido as técnicas de RI utilizadas.

2.4.5 LABRADOR

A abordagem do sistema Labrador [Mesquita et al. 2007] se difere das anteriores em três aspectos. Primeiro os autores adotaram redes Bayesianas pra estruturar as consultas e classificar os resultados, levando em consideração evidências relacionadas com o conteúdo do banco de dados. Ao fazer isso, utilizaram técnicas de recuperação da informação para medir a similaridade entre os termos de consultas e valores do banco de dados, que claramente apresentam melhor qualidade de resultado do que considerado apenas evidências baseadas em estruturas. Diferente da abordagem apresentada no DISCOVER, o LABRADOR é independente das características específicas dos SGBDs, como Oracle Text e IBM DB2 Text Information Extender, para processar consultas não-estruturadas. Isso permite busca em vários bancos de dados administrados por diferentes SGBDs com uma única consulta.

Segundo, foram utilizadas junções naturais para recuperar dados de múltiplas tabelas em vez de uma solução baseada *schema graph*. Este último restringe o administrador de banco de dados quando este pública um banco de dados, porque podem existir Restrições de Integridade Referencial (RIR) que não podem ser implementadas utilizando chaves estrangeiras. Além disso, nessa abordagem o administrador de banco de dados pode facilmente redefinir as junções naturais entre as tabelas originais publicando visões destas tabelas com diferentes nomes de atributos. Tal flexibilidade é crítica para a tarefa de publicação, em que o administrador deve ser capaz de definir as condições de junção independentemente de qualquer questão física de banco de dados.

Terceiro, o processamento da consulta não requer índices *full text* para os

atributos publicados. Assim, o Labrador pode publicar qualquer banco de dados sem mudanças no esquema ou nas instâncias dos dados.

2.4.6 SQLSUGG

Em *Interactive SQL Query Suggestion: Making Databases User-Friendly* [Fan, Li e Zhou 2011] os autores propõem uma forma de ajudar os usuários, já com alguma experiência em linguagem SQL, a criar consultas SQL que retornem as tuplas a partir de uma busca que contém as palavras-chave. Diferente dos demais, não está preocupado em retornar dados relevantes para o usuário leigo, e sim retornar consultas SQL que serão executadas pelo usuário afim de ajudá-lo a ganhar tempo e torná-lo mais produtivo, e estas sim, retornam os dados mais relevantes.

Assim como nos outros trabalhos, os autores utilizam técnicas de RI para classificar as consultas geradas para entregar repostas de maior relevância e adotam a abordagem de *Data Graph* para conectar as tuplas baseado nas palavras-chave fornecidas pelo usuário.

2.4.7 CSTree

O trabalho *"Providing built-in keyword search capabilities in RDBMS"* [Li et al. 2010] é o que mais se assemelha com o que propomos nesta dissertação. Também propõe integrar no banco de dados a capacidade de busca por palavras-chave, porém, com algumas diferenças que são fundamentais:

- Utiliza a abordagem *Data Graph*. No mecanismo que implementamos é utilizada a abordagem de *Schema Graph* e um algoritmo baseado em *Steiner Trees* para gerar as *Candidate Networks*. Acreditamos que a abordagem utilizada no Lathe [Oliveira e Silva 2015,Oliveira e Silva 2017] é mais eficiente uma vez que número de árvores geradas baseadas no grafo do esquema do banco de dados é bem menor que o número de árvores baseado no grafo de dados.
- Os autores apresentam um novo conceito chamado *Compact Steiner Tree* (CSTree), que é um algoritmo apresentado como solução aproximada para o problema das árvores de Steiner, utilizado para responder quais são as top-K respostas para a consulta, por pavalras-chave, fornecida pelo usuário. Nós utilizamos o bem conhecido algoritmo de Takahashi [Takahashi e Matsuyama 1980] para calcular as árvores mínimas de Steiner.

3 Lathe

De forma a manter esta dissertação auto-contida, nesse capítulo apresentamos uma visão geral do sistema Lathe [Oliveira e Silva 2015, Oliveira e Silva 2017] que é a base para o nosso trabalho. O Lathe se encaixa na categoria de sistemas baseados em *Schema Graph*, porém inova ao adotar a abordagem de *Steiner Trees* para o problema da geração de CNs.

3.1 Visão geral

A arquitetura do sistema Lathe está ilustrada na Figura 5. Dada uma consulta por palavras-chave o Lathe, como todos os outros sistemas R-KwS em sua categoria, gera uma série de *Candidate Networks* e então executa essas CNs para produzir as respostas para a consulta do usuário. A maior diferença entre o Lathe e todos os outros sistemas R-KwS está na forma como são geradas essas CNs. Até então, todos os sistemas na literatura usam o algoritmo *CNGen*, proposto no sistema DISCOVER [Hristidis e Papakonstantinou 2002]. Em alguns casos, existem passos adicionais para melhorar o conjunto de CNs geradas, isso ocorre nos sistemas KwS-F [Baid et al. 2010] e CNRank [Oliveira, Silva e Moura 2015].

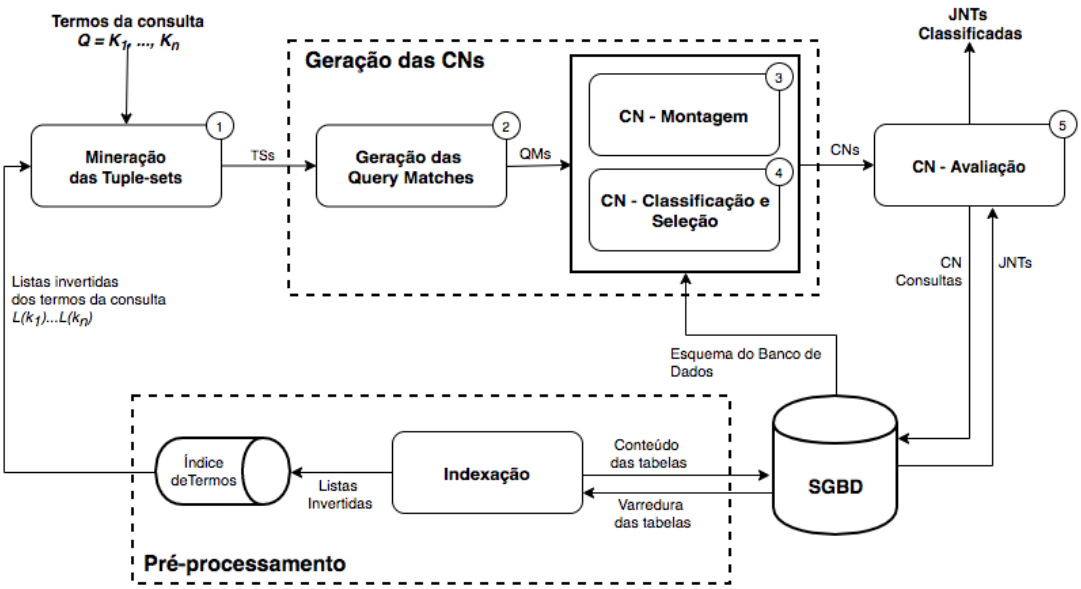


Figura 5 – Lathe: Arquitetura e principais funcionalidades.

Como já mencionamos, CNs são expressões da álgebra relacional que fazem a junção de tabelas cujas as tuplas contêm as palavras-chave procuradas, ou seja, cada CN dá origem a uma consulta SQL cuja semântica é equivalente a da consulta original e produz, como resultado, árvores de tuplas que satisfazem a consulta original do usuário.

Por exemplo, considere a consulta *'denzel washington gangster'* e suponha que queiramos executar essa consulta em um SGBD que contenha o banco de dados IMDb [Coffman e Weaver 2010], que é um banco de dados bem conhecido de filmes chamado *Internet Movie Databases*. Uma possível CN para essa consulta seria dada por:

$$\sigma_{\text{title}\supseteq\{gangster\}}\text{TITLE} \underset{\text{id=movie\_id}}{\bowtie} \text{CAST\_INFO} \underset{\text{person\_id=id}}{\bowtie} \sigma_{\text{name}\supseteq\{denzel\ washington\}}\text{NAME} \tag{3.1}$$

Onde NAME é a tabela que armazena os nomes das pessoas envolvidas (atores, atrizes, diretores e etc.), TITLE armazena os títulos dos filmes, e CAST_INFO associa os filmes com as pessoas que as estão envolvidas nos filmes. As junções desta expressão são derivadas das Restrições de Integridade Referêncial (RIRs) do banco de dados.

A complexidade na geração de CNs está principalmente em dois fatores:

- podem existir múltiplos *Tuple-Sets* para cada subconjunto dos termos da consulta. Como consequência, pode existir um número muito grande de maneiras de combinar esses *Tuple-Sets* para que todos os termos da consultas sejam cobertos;
- dado um conjunto de *Tuple-Sets*, que cobrem os termos da consulta, existem muitas maneiras distintas de conectá-los através de suas RIRs e *Free Tuple-Sets*;

O Lathe adota uma nova abordagem para a geração de CNs, que detalhamos a seguir.

3.1.1 Mineração de *Tuple-Sets*

No Lathe, quando a consulta por palavras-chave é submetida pelo usuário para processamento, o primeiro passo é identificar os *Tuple-Sets* que podem potencialmente ser utilizados para gerar as *Cantidate Networks*, passo ① da Figura 5. A geração dos *Tuple-Sets* começa determinando quais tabelas e tuplas contêm as palavras-chave da consulta. Esta informação é obtida diretamente de uma estrutura chamada *Índice dos termos*. O Índice dos termos é uma lista invertida que, além do termo, contém 3 informações básicas associadas a cada termo:

- o atributo, juntamente com a tabela, onde o termo ocorre;

- a frequência com que ocorre;
- os IDs que identificam cada tupla onde o termo ocorre na tabela;

O índice dos termos é construído em uma etapa de pré-processamento onde todas as tabelas são analisadas e a lista invertida é criada.

Computando as interseções não vazias das listas invertidas descritas acima, podemos encontrar todas tuplas das tabelas do banco de dados que contêm qualquer subconjuntos dos termos da consulta, ou *(termsets)*. No Lathe, as interseções são computadas em memória utilizando um algoritmo que foi desenvolvido baseado no algoritmo ECLAT [Zaki 2000]. Esse algoritmo, que foi chamado de TSFind, minera *termsets* de tamanho crescente, a partir de 1 elemento, e então realiza múltiplas intersecções entre os *termsets*, gerando conjuntos de tamanho crescente até não ser possível obter novas intersecções.

Por exemplo, na Figura 6 são representadas as duas partes do processo de mineração de *Tuple-Sets*. Os retângulos representam os *tuple-sets* onde, na parte superior de cada retângulo estão os *termsets* e, logo abaixo, os IDs das tuplas onde os *termsets* ocorrem. Neste exemplo foi submetida, para o algoritmo, a consulta $Q = \{denzel, washington, gangster\}$. Na Parte 1 do processo, é gerada uma lista de *tuple-sets* para *termsets* de tamanho 1 e, no nosso exemplo, são gerados 3 *tuple-sets* correspondendo a cada um dos termos da consulta Q. Logo ao lado estão representadas as iterações que ocorrem na Parte 2 do processo. Nessa parte são computadas todas as intersecções entre as ocorrências dos *termsets*, obtidos até então, e são construídos novos *termsets* com as intersecções não vazias de suas ocorrências. Como podemos observar, a intersecção das ocorrências dos *termsets {denzel}* e *{gangster}*, na Iteração 1, cria um subconjunto de ocorrências não vazio *{c1}*, gerando assim um novo *termset*, *{denzel, gangster}*. Após a criação do novo *termset* as ocorrências da intersecção são removidas dos *tuple-sets* originais. Este processo se repete até que não seja encontrada nenhuma nova intersecção não vazia. Nesse ponto pode ocorrer que algum *termset* fique com um conjunto vazio de ocorrências, como podemos observar na Iteração 2 o *termset {denzel, gangster}*, que ficou com um conjunto vazio de ocorrências com a criação do *termset {denzel, gangster, washington}*, nesses casos os *termsets* com um conjunto vazio de ocorrências são descartados. Finalmente, o conjunto de *tuple-sets* gerados para a consulta $Q = \{denzel, washington, gangster\}$, ao final da iteração 3, é: {{denzel, c2},{gangster, c5}, {washington, c3}, {{denzel, gangster, washington},c1}}.

A forma como o Lathe realiza a tarefa de minerar *tuple-sets* é a maior diferença em comparação ao que é feito em todos os sistemas baseados no DISCOVER. No DISCOVER um módulo chamado *Tuple Set Post-Processor* materializa os *tuple-sets* como tabelas no banco de dados executando muitos comandos INTERSECT.

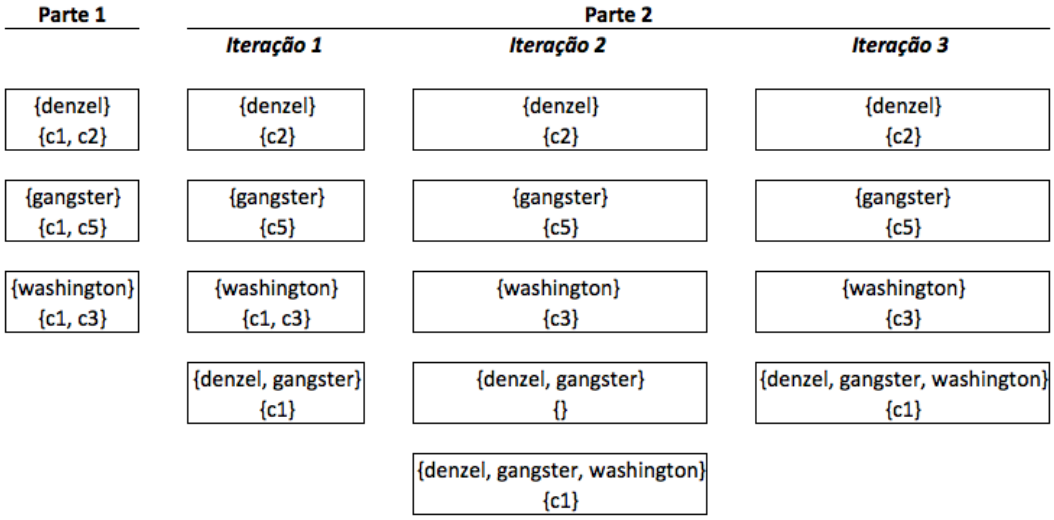


Figura 6 – Passos realizados pelo algoritmo de mineração de *tuple-sets*.

3.1.2 Geração das *Query Matches*

Na caracterização do problema de *Geração de Candidate Networks*, os autores do Lathe apresentaram um novo conceito, chamado de correspondência - *match*, e provam uma importante propriedade:

Seja C uma *joining network of tuple-sets* e $M = \{R_1^{k_1}, \ldots, R_m^{k_m}\}$ o seu conjunto de *non-free tuple-sets*. Se C é uma *candidate network* para uma consulta Q, então, $K_1, K_2, \ldots, K_m$ deve formar o conjunto de cobertura - *set covering* para o conjunto de palavras-chave de Q, e cada $R_i^{K_i} \in M$ deve ser um *non-free tuple set*. Nesse caso, M é chamado de *match* para Q.

Dada uma CN como a da Equação 3.1, o conjunto dos *non-free tuple-sets* é uma consulta correspondente - *Query Macth*. No sistema Lathe, é adotada uma abordagem *botton-up* para obter as CNs, na qual primeiramente são geradas os *Query Macthes* e então estes são usados para montar as CNs, passo ② na Figura 5.

O seguinte exemplo é baseado no banco de dados do IMDb (ver Figura 3). Considere novamente a consulta $Q = \{denzel, washington, gangster\}$. Alguns possíveis *set coverings* para Q são:

$V_1 = \{\{denzel\}, \{washington\}, \{gangster\}\}$,

$V_2 = \{\{denzel, washington\}, \{gangster\}\}$,

$V_3 = \{\{denzel, gangster\}, \{washington\}\}$, e assim por diante.

Agora considere o *set covering* V_2 cujos os subconjuntos são $\{denzel, washington\}$ e $\{gangster\}$. Suponha que o termo *"gangster"* ocorra sozinho nas tuplas das tabelas CHAR, MOVIE e CAST. Os *tuple-sets* para este termo são:

$T_1 = CHAR^{\{gangster\}}$

$T_2 = MOVIE^{\{gangster\}}$

$T_3 = CAST^{\{gangster\}}$

Também, suponha que os termos *"denzel"* e *"washington"* ocorram juntos, nas tuplas das tabelas NAME e CAST. Então, os *tuple-sets* para esses termos são:

$T_4 = NAME^{\{denzel,washington\}}$

$T_5 = CAST^{\{denzel,washington\}}$

Considerando os subconjuntos do *set covering* V_2, alguns dos possíveis *matches* são:

$M_1 = \{CHAR^{\{gangster\}}, NAME^{\{denzel,washington\}}\}$

$M_2 = \{CHAR^{\{gangster\}}, CAST^{\{denzel,washington\}}\}$

$M_3 = \{MOVIE^{\{gangster\}}, NAME^{\{denzel,washington\}}\}$

$M_4 = \{MOVIE^{\{gangster\}}, CAST^{\{denzel,washington\}}\}$

$M_5 = \{CHAR^{\{gangster\}}, CAST^{\{denzel,washington\}}\}$

$M_6 = \{CAST^{\{gangster\}}, CAST^{\{denzel,washington\}}\}$

...

Note que, para esses 3 termos, mais de 100 *set coverings* distintos serão gerados. No caso de $V_1 = \{\{denzel\}, \{washington\}, \{gangster\}\}$, apresentado acima, 19 *set coverings* distintos são gerados. Assim, podem existir muitos *matches*, mas isso depende da distribuição dos termos da consulta entre as tabelas do banco de dados.

Intuitivamente, cada *match* representa uma forma diferente de combinar tuplas contendo os *termsets*. É assumido que as respostas devem conter todos os termos da consulta, e isso quer dizer que cada termo da consulta deve aparecer em pelo menos um *tuple set* em uma CN. Assim, a união de todos os termos utilizados nos predicados de uma consulta formam o *string convering* da consulta.

No exemplo da Equação 3.1, uma *query match* é dada por:

$$\{ \sigma_{\substack{\text{TITLE} \\ \text{title} \supseteq \{gangster\}}}, \sigma_{\substack{\text{NAME} \\ \text{name} \supseteq \{denzel,washington\}}} \} \tag{3.2}$$

Nesse caso, temos o seguinte *string covering* para a consulta 'denzel washington gangster': *{{gangster}, {denzel, washington}}*.

Observe que muitas combinações de *tuple-sets* podem gerar, consequentemente, muitos *string coverings*, conforme os termos da consulta estão distribuídos entre as tabelas do banco de dados. Assim, muitos *macthes* podem existir e muitas CNs podem

ser "montadas", ligando os *tuple-sets* de um *match*.

3.1.3 Obtendo as *Query Matches*

Com isso, cada *match* para a consulta Q é um conjunto de *non-free tuple-sets* na forma $M(Q) = \{R_1^{k_1}, \ldots, R_m^{k_m}\}$, onde existe algum *set covering* $V = \{k_1, \ldots, K_m\}$ para o conjunto de palavras-chave em Q, e todo $R_j^{K_j} \neq \emptyset (1 \geq j \geq m)$. Seja $M(Q)$ o conjunto de todos os *query matches* para Q dada a instância do banco de dados I. O tamanho de $M(Q)$ depende de dois fatores. O primeiro é o número de n de *set coverings*. O segundo fator é como os subconjuntos dos termos da consulta são distribuídos entre as tabelas do banco de dados, isto é, o número de *tuple-sets* para os subconjuntos dos termos da consulta.

O primeiro fator depende do número de termos na consulta. Infelizmente, n cresce exponencialmente conforme o número de termos da consulta [Macula 1994]. Esta é uma preocupação, mesmo considerando que o número de termos na consulta geralmente é pequeno, geralmente menos que 4 e frequentemente em torno de 2. Por exemplo, enquanto que para 4 termos nós temos cerca de 32000 possíveis *coverings*, para 5 termos existem cerca de 2 bilhões de possíveis *coverings*. Assim, gerar todos os *coverings* possíveis para cada consulta é inviável, embora seja formalmente necessário gerar todas as combinações de subconjuntos dos termos da consulta.

Por outro lado, o segundo fator é mais difícil de prever, mas geralmente poucos subconjuntos de termos da consulta são frequentes em muitas tabelas. De fato, subconjuntos maiores são menos frequentes.

Com base nessas observações, é evidente que, embora seja formalmente requerido, não é viável, e nem útil, gerar todos os *set coverings* para obter todos os *matches* da consulta. Dessa forma, foram aplicada algumas heurística para podar o maior número possível de *matches* sem comprometer a qualidade dos resultados. De fato, estratégias de poda são necessárias para fazer a geração de *Candidate Networks* um processo viável. Tais estratégias são aplicadas, por exemplo, no algoritmo CNGen, utilizado no sistema DISCOVER [Hristidis e Papakonstantinou 2002].

No sistema Lathe essas ideias são usadas para separar a geração de CNs em dois passos. Primeiro, o sistema combina os *tuple-sets* recuperados para formar os *matches*. Então, processando o esquema do banco de dados, modelado como um grafo, procura formas de conectar as *tuple-sets* e cada *match* e monta as CNs. Esta abordagem para obter as CNs é completamente nova em comparação com o algoritmo CNGen usado no sistema DISCOVER e todo os outros sistemas R-KwS baseados em *Schema Graph* até agora.

3.1.4 CN - Montagem

No passo ③ da Figura 5, cada *Query Match* selecionado é usado para gerar CNs. O problema aqui está em como conectar os *tuple-sets* que formam os *Query Matches* através de um ou mais *free tuple-sets*, ou seja tabelas no grafo do esquema do banco de dados que criam caminhos entre os *tuple-sets*. Considere momentaneamente um caso simples onde um dado *match* envolve dois *tuple-sets*. Para conectá-los é possível enumerar todos os caminhos possíveis entre esses *tuple-sets* percorrendo o grafo do esquema do banco de dados. Assim cada caminho corresponde a uma CN. Para reduzir o número de CNs geradas, podemos gerar somente uma CN, correspondente ao *caminho mínimo* entre dois *tuple-sets*. Nesse caso somente uma CN, a menor de todas, é gerada. A generalização desse problema, para *matches* que incluem mais de dois *tuple-sets* pode ser modelado como o problema das *Steiner Trees*.

Embora o uso do conceito de *Steiner Trees*, para ligar as tuplas que contém as palavras-chave seja comum em sistemas R-KwS baseados em *Data Graphs*, essa abordagem não foi considerada antes para o problema da geração das *Candidate Networks*, que são a base para sistemas da categoria *Schema Graphs* sendo essa uma forma inovadora de gerar as CNs que foi adotada no Lathe. No Lathe foi implementado a montagem de CN utilizando o bem conhecido algoritmo de Takahashi [Takahashi e Matsuyama 1980] para calcular as árvores mínimas de Steiner.

O problema das *Steiner Trees* consiste em dado um conjunto de pontos, denominados terminais, prova-se que existe um grafo minimal que os conectas, chamado *Steiner Tree*. Esses terminais podem representar circuitos elétricos, servidores em uma rede de computadores e, trazendo a ideia para o nosso trabalho, a ligação entre as tabelas de um banco de dados através de suas restrições de integridade referencial. Um modo de visualizar as *Steiner trees* é com o experimento com sabão: em uma solução de água e sabão, ao mergulhar duas placas paralelas ligadas por pinos e em seguida remover, uma película de sabão irá se formar conectando os pinos, conforme demonstrado da Figura 7. Como já é sabido as películas de sabão realizam as superfícies mínimas.

Essas películas formam as arestas de um grafo, onde os além dos vértices formados pelos pinos nas placas, também surgem outros vértices no encontro das películas. Esses novos vértices são chamados de *Steiner Nodes*. As árvores geradas nessa experiência têm a propriedade de serem soluções locais do Problema das *Steiner Trees*.

No sistema DISCOVER [Hristidis e Papakonstantinou 2002], o grafo G_{TS} é definido como um grafo cujos os vértices são todos os *tuple-sets* $R_i^{K_i}$, incluindo os *free tuple-sets*, e existe uma aresta $\langle R_i^{K_i}, R_j^{K_j} \rangle$ em G_{TS} se o *schema graph* G_u tem a aresta $\langle R_i, R_j \rangle$. Claramente cada *tuple set* em cada *match* é um vértice em G_{TS}. Assim, CNs são árvores (i.e. subgrafos) de G_{TS} que incluem *free tuple-sets* e *non-free-tuple-sets* que

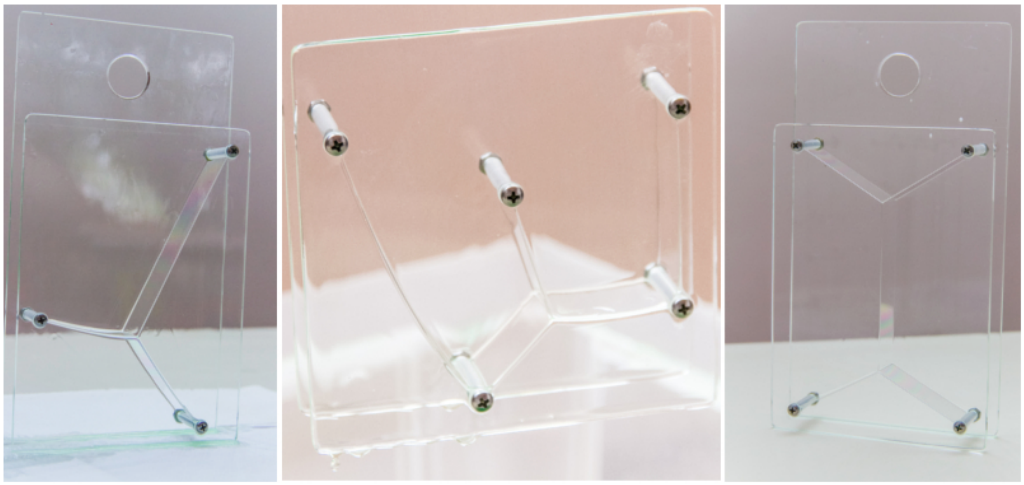


Figura 7 – *Steiner Trees* geradas com película de sabão.

compõem o *match*. Note que dado um *match* em G_{TS} podem ocorrer diferentes CNs, porque podem existir muitas maneiras de conectar os *non-free tuple-sets* através dos *free tuple-sets* em G_{TS}. A seguir, usamos essa abordagem para caracterizar as CNs.

Definição 7: Seja G_{TS} um grafo do esquema para a consulta Q, conforme definido no sistema DISCOVER. Dado uma combinação - *match* M de Q, definimos um *matching graph* para M, denotado por $G_{TS}[M]$ que é um subgrafo de G_{TS} cujos os nós são todos *free tuple-sets* de G_{TS} incluindo os *non-free tuple-sets* que compõem M.

Dado um grafo correspondente - *matching graph* $G_{TS}[M]$, as CNs que contém o *match* M podem ser definidas nos termos do problema das *Steiner Trees*. Lembrando que, dado um grafo não direcionado $G\langle V, E\rangle$ e o conjunto de arestas $N \subset V$, uma *Steiner Tree* é qualquer árvore T em G que contém todos os vértices de N, que são chamados de vértices terminais. De fato, cada CN que usa os *non-free tuple-sets* em M é uma *Steiner Tree* extraída de G_{TS} onde cada *non-free tuple set* é um nó terminal e os *free tuple-sets* são os *Steiner Nodes*, ou seja, nós não terminais que conectam os terminais.

3.1.5 CN - Classificação e Seleção

Dependendo da quantidade de termos da consulta, da disposição destes termos nas tabelas do banco de dados e de seu tamanho, podem existir muitas *Candidates Networks* e processar um número grande de CNs demanda tempo e consome recursos.

Por outro lado, apenas um pequeno número de CNs produzem respostas relevantes para o usuário e somente essas, de fato, merecem ser processadas. Esta afirmação esta em sintonia com as feitas por outros pesquisadores [Luo et al. 2007, Nandi e Jagadish 2009, Baid et al. 2010, Coffman e Weaver 2010], que descobriram que o número de respostas relevantes para consultas por palavras-chave é, muitas vezes,

pequeno e que em muitos casos existe apenas uma resposta relevante para retornar.

Dessa forma, o processo de classificação das CNs e fundamental para tornar o processo de busca por palavras-chave viável.

No sistema CNRank [Oliveira, Silva e Moura 2015] foi demonstrado que não é necessário processar todas a CNs geradas, por que, de fato, somente poucas CNs produzem respostas relevantes para a consulta do usuário. Foi então proposto um algoritmo para classificar e selecionar somente algumas CNs que serão avaliadas, isto resultou em uma drástica redução do tempo que se leva para avaliar as CNs sem comprometer a qualidade do resultado. Assim, uma vez que as CNs tenham sido geradas, estas são classificadas e somente algumas do topo da lista são selecionadas para a avaliação. Esta etapa corresponde ao passo ④ da Figura 5.

Para classificar as CNs é atribuído um peso para cada uma, que é calculado aplicando o conceito de *Term Frequency (TF)* e *Inverse Attribute Frequency (IAF)* [Mesquita et al. 2007].

O *TF* mede a frequência de um dado termo nos valores de um atributo, considerando todas as tuplas da tabela onde este ocorre. E e calculado pela seguinte formula:

$$tf = \frac{log(1 + Fkj)}{log(1 + Nj)}$$

Onde *Fkj* é o número de ocorrências de cada termo em cada um dos atributos do banco de dados e *Nj* é o número total de termos distintos que ocorrem em cada atributo textual.

Já o *IAF* é uma adaptação do conceito de *Inverse Document Frequency (IDF)* que é utilizado no contexto de Recuperação da Informação [Baeza-Yates e Ribeiro-Neto 2011]. O IAF calcula quanto frequente é o termo em relação aos atributos onde este ocorre, de acordo com a seguinte formula:

$$iaf = log\left(1 + \frac{Na}{Ck}\right)$$

Onde *Na* é o número total de atributos no banco de dados e *Ck* é total de atributos onde cada termo ocorre. O IAF é utilizado como uma estimativa do grau de ambiguidade do termo em relação aos atributos do banco de dados. Os pesos de acordo com o modelo TF-IAF é calculado pela seguinte fórmula.

$$w_{fk} = tf \times iaf = \frac{log(1 + Fkj)}{log(1 + Nj)} \times log\left(1 + \frac{Na}{Ck}\right)$$

3.1.6 CN - Avaliação

Para a avaliação das CNs ⑤, o Lathe adota o algoritmo *Skyline Sweeping* usado no sistema SPARK [Luo et al. 2007]. Nos experimentos os autores destacam que obtiveram resultados duas vezes mais precisos em comparação com os resultados quando é usado o algoritmo tradicional CNGen, que também é adotado no SPARK.

Nesta dissertação implementamos o sistema Lathe até o passo ④. Nosso principal objetivo, que é o de embutir um mecanismo de busca por palavras-chave em um SGDB foi alcançado e geramos *Candidates Networks* de qualidade, como mostram os experimentos, e, diferente dos demais trabalhos, que até então eram feitos em ambientes externos ao SGBD, criamos uma ferramenta flexível que pode ser facilmente utilizada por desenvolvedores em suas aplicações, dando a estes a liberdade de utilizar um mecanismo de busca por palavras-chave em qualquer sistema que use um SGDB.

4 Lathe-DB

Neste capítulo apresentamos o nosso mecanismo de consultas por palavras-chave, chamado Lathe-DB, que foi desenvolvido embutido no próprio ambiente do SGBD. Nós começamos apresentando uma visão geral e a arquitetura do nosso mecanismo e, em seguida, detalhamos cada etapa do funcionamento do mesmo.

4.1 Visão geral

Ao desenvolver um sistema R-KwS embutido em um SGBD, podemos tirar proveito dos recursos e robustez tipicamente oferecidos por ele. Isto difere da maioria dos trabalhos propostos até então na literatura, onde os sistema R-KwS são desenvolvidos fora do ambiente do SGBD.

Nossa implementação do Lathe-DB foi realizada considerando uma instalação padrão do SGBD PostgreSQL, sem alterar seu código-fonte e sem lançar mão do uso de módulos adicionais que ampliassem seu funcionamento. Dessa forma, utilizamos somente os recursos comumente disponibilizados pelo SGBD, como tabelas, índices e funções, com o intuito de facilitar a sua portabilidade. Assim, desenvolvemos uma solução prática que disponibiliza a busca por palavras-chave utilizando uma simples consulta SQL e que pode facilmente ser aplicada em bancos de dados dos mais variados domínios.

4.2 Arquitetura

A arquitetura do Lathe-DB está ilustrada na Figura 8, e mostra que, diferente da arquitetura do Lathe [Oliveira e Silva 2015, Oliveira e Silva 2017] apresentada no Capítulo 3, não há acesso externo ao SGBD, uma vez que todas as funcionalidades do Lathe-DB já estão embutidas no próprio SGBD.

Na arquitetura do Lathe-DB, assim como no Lathe, podemos observar duas fases distintas, a de **Pré-Processamento**, que prepara o banco de dados para consultas por palavras-chave, e a de **Geração das CNs**, que é composta pelos passos de Mineração de *Tuple-Sets* ①, Geração das *Query Matches* ②, Classificação das *Candidate Networks* ③ e Montagem das *Candidate Networks* ④.

Originalmente, o sistema Lathe foi escrito em Java e trabalha em uma infraestrutura paralela ao SGBD. Dessa forma, conta com todos os recursos da linguagem Java, como o uso de estruturas de dados sofisticadas em memória, tais como: pilhas, filas, listas e

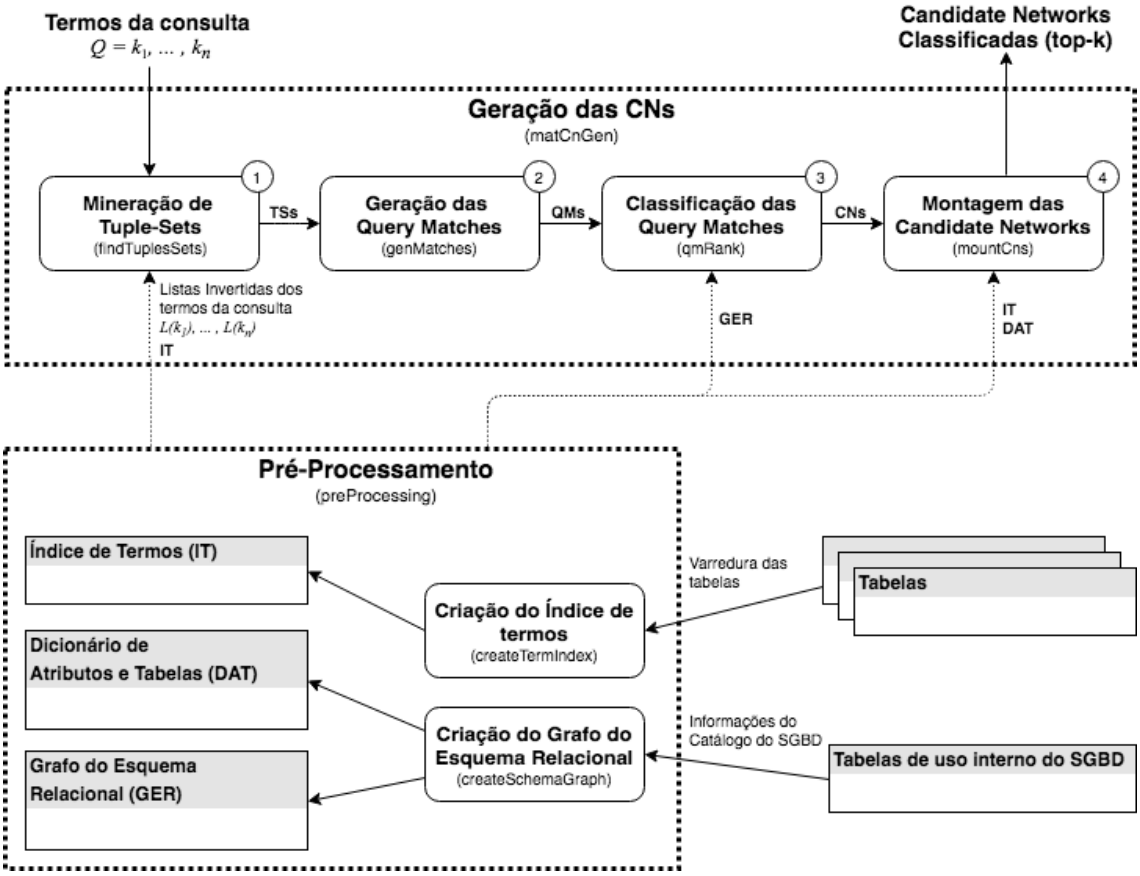


Figura 8 – Arquitetura do Lathe-DB.

etc. Os desafios encontrados para contornar as limitações impostas pelo uso somente da linguagem PL/pgSQL são descritas ao longo das próximas seções.

4.3 Pré-Processamento

Nesta seção apresentamos o algoritmo utilizado na fase de pré-processamento, chamado ***preProcessing***. Este algoritmo recebe como parâmetro opcional de entrada o nome do esquema[1] e realiza três operações distintas, criando:

- O *Índice de Termos*, que conceitualmente, é uma lista invertida de termos com a localização das ocorrências desses termos nos atributos textuais nas tabelas do banco de dados. Além disso, são armazenados alguns valores pré-calculados que serão utilizados na Classificação das *Candidate Networks* (passo ④ da Figura 8).

[1] Caso não seja informado o nome do esquema, o algoritmo *preProcessing* vai, por padrão, indexar o esquema *public*.

- O *Dicionário de Atributos e Tabelas*, que será utilizado para construir a *Grafo do Esquema Relacional* do banco de dados que auxilia na Geração das *Candidate Networks*.

- O *Grafo do Esquema Relacional*, que também será utilizado no processo de Geração das *Candidate Networks*.

Para um melhor entendimento, a Figura 9 mostra a arquitetura da implementação do algoritmo, sua interação com os algoritmos de apoio e com as tabelas do banco de dados. Todos esses passos serão explicados em detalhes mais adiante nesse capítulo.

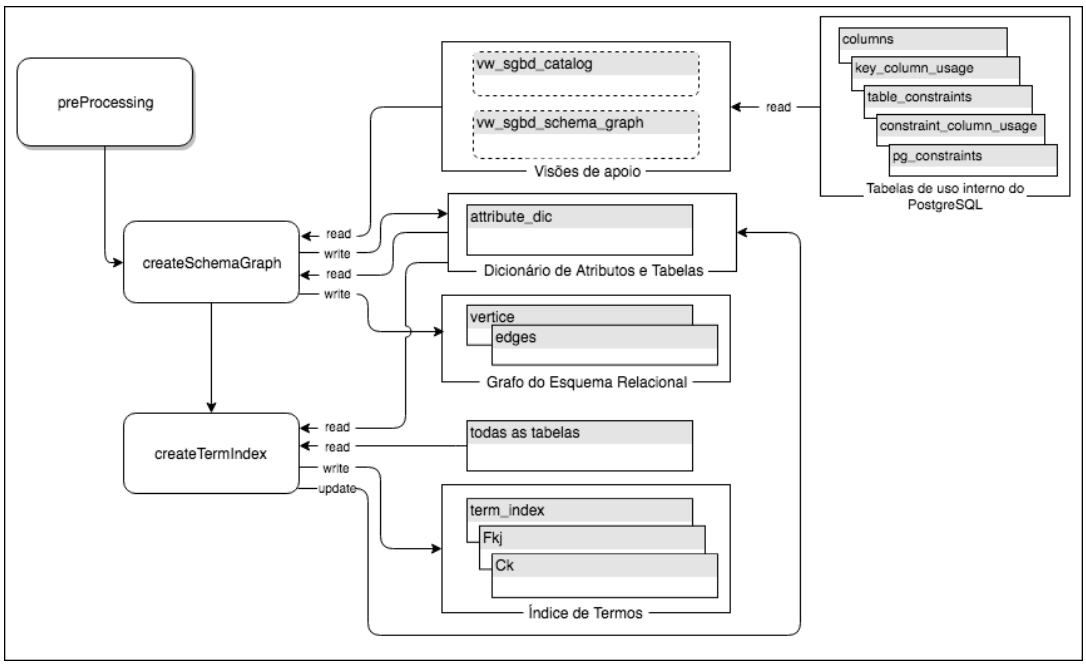


Figura 9 – Arquitetura do Algoritmo *preProcessing*.

4.3.1 Índice de Termos

Como podemos observar na Figura 10, o Índice de Termos (IT) é composto por três tabelas:

- ***term_index:*** é a principal tabela, onde são armazenados os termos que ocorrem no banco de dados e um array do tipo *occurrence* que contém a localização dos termos nas tabelas, criando um índice invertido que indexa os termos que ocorrem em atributos textuais, ou seja, atributos dos tipos: char, varchar, text, etc.

- ***Fkj:*** é a tabela que armazena o número de ocorrências de cada termo em cada atributo.

- ***Ck:*** tabela que armazena o total de atributos onde o termo ocorre.

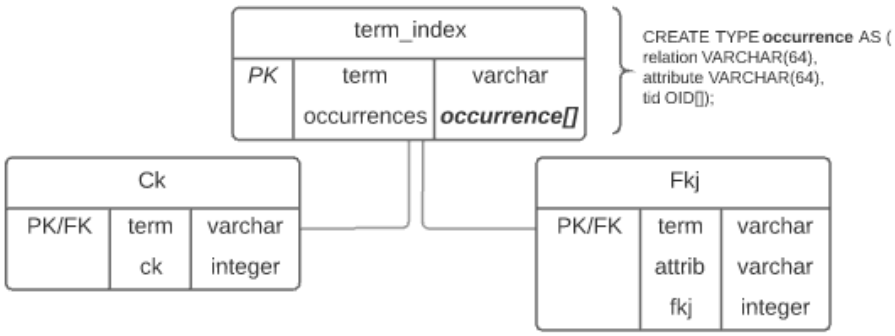


Figura 10 – Tabelas que compõem o Índice de Termos (IT).

Para contornar a falta de estruturas de dados mais sofisticadas, na tabela *term_index* usamos um atributo, chamado *occurrence*, que é um array de um tipo composto, e este, por sua vez, contém três atributos, que armazenam os nomes da tabela e do atributo onde o termo ocorre, e um array do tipo OID, que armazena a lista de tuplas onde o termo ocorre, simulando assim uma lista invertida. O tipo OID é disponibilizado pelo PostgreSQL para armazenar identificadores de objetos, do inglês *Object IDentifiers (OID)* que são atributos ocultos mantidos pelo SGBD que identificam unicamente cada tupla.

Os valores contidos nas tabelas ***Fkj*** e ***Ck*** são utilizados pelo algoritmo de Classificação das *Query Matches*, que será explicado mais adiante neste capítulo. Embora os dados contidos nestas tabelas pudessem ser derivados da tabela *term_index*, nós optamos por materializá-los por conveniência para facilitar o processo de Classificação das *Query Matches* (ver Seção 4.4.3).

4.3.2 Dicionário de Atributos e Tabelas

Na etapa de pré-processamento também é populada uma tabela, chamada *attribute_dic*, que contém o Dicionário de Atributos e Tabelas (DAT), ver Figura 11. A informação armazenada nessa tabela é obtida a partir do catálogo do banco que dados, que são tabelas mantidas pelo SGBD que contêm os dados de todos os objetos do banco de dados, tais como atributos, tabelas, nomes do esquema e dados das Restrições de Integridade Referencial (RIR). Um destaque nessa tabela é o atributo *Nj*, que armazena, para cada atributo textual, o número de termos distintos que nele ocorre. Essa informação, assim como as contidas nas tabelas *Ck* e *Fkj*, é utilizada pela função de classificação de *Query Matches*. Apesar de ser possível obter esses dados a partir do catálogo do banco de dados, optamos também por materializar esses dados por conveniência para facilitar o processo de Montagem das CNs (ver Seção 4.4.4).

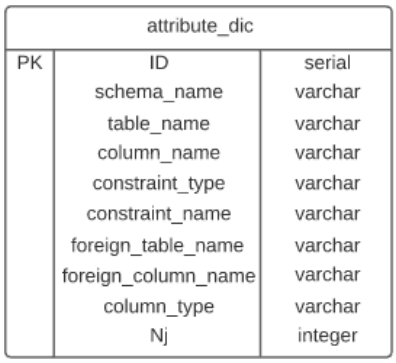


Figura 11 – Tabela que armazena o Dicionário de Atributos e Tabelas (DAT).

4.3.3 Grafo do Esquema Relacional

Ainda na fase de pré-processamento também é persistido o Grafo do Esquema Relacional (GER), que é armazenado na forma de listas de adjacências e é criado utilizando as informações contidas no DAT. Conforme podemos observar na Figura 12, o GER é uma estrutura composta por duas tabelas:

- ***vertices:*** tabela onde são armazenados os vértices que compõem o grafo. Na nossa implementação o atributo *vertice* recebe os nomes das tabelas do banco de dados.

- ***edges:*** onde são armazenadas as arestas do grafo. Os atributos *source* e *destination*, armazenam os vértices de origem e destino respectivamente, em outras palavras, a tabela de origem e a relação de destino. E finalmente, o atributo *join_predicate* armazena o predicado usado para a junção entre a tabela de origem e a tabela de destino, ou seja, a sua Restrição de Integridade Referencial (RIR), na forma $\langle table\_source.attrib = table\_destination.attrib \rangle$. O atributo *join_predicate* será utilizado na construção das *Candidate Networks* na etapa de Montagem das CNs.

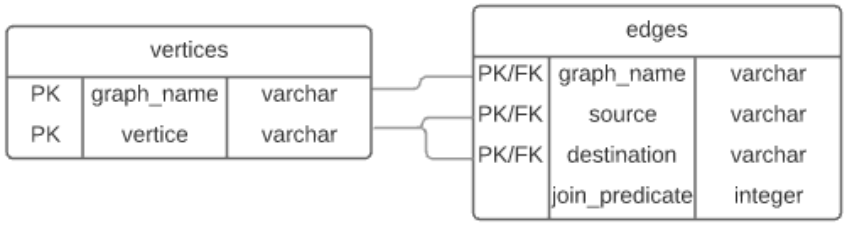
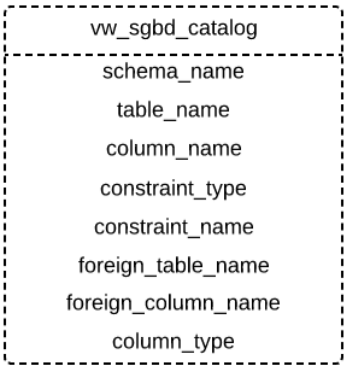


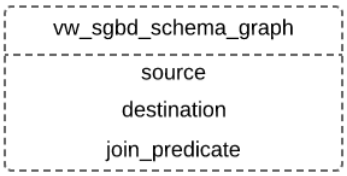
Figura 12 – Tabelas que armazenam o Grafo do Esquema Relacional (GER).

4.3.4 Algoritmo

Para otimizar a escrita do algoritmo *preProcessing* criamos as visões *vw_sgbd_catalog* e *vw_sgbd_schema_graph*, ver Figura 13. No caso do PostgreSQL, essas visões são criadas a partir das tabelas de uso interno no SGBD: *columns*, *key_column_usage*, *table_constraints*, *constraint_column_usage* e *pg_constraints*, ver Figura 14.




```
CREATE VIEW lathedb.vw_sgbd_catalog as
SELECT
  c.table_schema AS schema_name,
  c.table_name,
  c.column_name,
  tc.constraint_type,
  kcu.constraint_name,
  ccu.table_name AS foreign_table_name,
  ccu.column_name AS foreign_column_name,
  c.udt_name AS column_type
FROM information_schema.columns c
LEFT JOIN information_schema.key_column_usage kcu ON c.table_catalog = kcu.table_catalog
  AND c.table_schema  = kcu.table_schema AND c.table_name   = kcu.table_name
  AND c.column_name   = kcu.column_name
LEFT JOIN information_schema.table_constraints tc ON kcu.table_catalog = tc.table_catalog
  AND kcu.table_schema  = tc.table_schema AND kcu.table_name   = tc.table_name
  AND kcu.constraint_name = tc.constraint_name
LEFT JOIN information_schema.constraint_column_usage AS ccu  ON
   ccu.constraint_name = tc.constraint_name AND tc.constraint_type  = 'FOREIGN KEY'
WHERE c.table_schema NOT IN ( 'information_schema', 'pg_catalog' )
```




```
CREATE VIEW lathedb.vw_sgbd_schema_graph as
SELECT DISTINCT
  d.table_name AS source,
  d.foreign_table_name AS destination,
  lathedb.getjoin(pg_catalog.pg_get_constraintdef(r.oid),
        d.table_name, d.foreign_table_name) AS join_predicate
FROM lathedb.attribute_dic d
JOIN pg_catalog.pg_constraint r ON d.constraint_name = r.conname
WHERE d.constraint_type = 'FOREIGN KEY' AND d.table_name != d.foreign_table_name
```

Figura 13 – Visões de apoio ao algoritmo *preProcessing* e ao lado os códigos SQL utilizados para a criação de ambas.

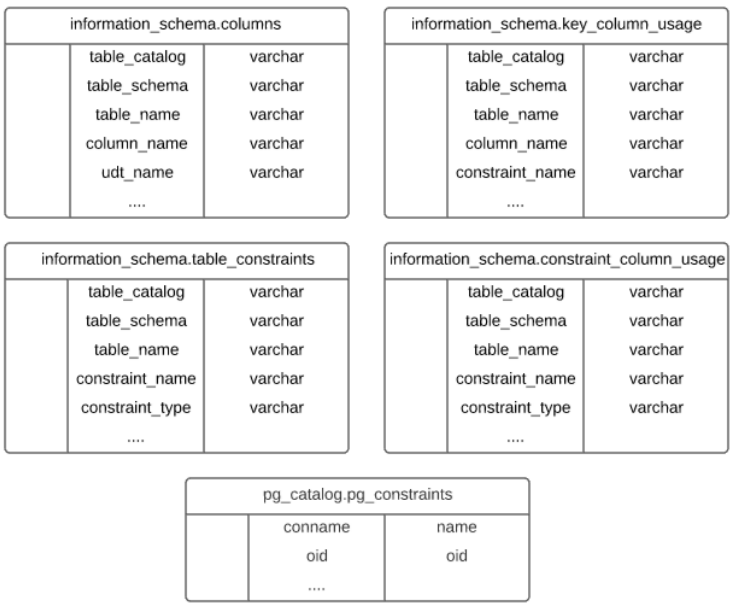


Figura 14 – Tabelas de uso interno do PostgreSQL.

As tabelas de uso interno, utilizadas nas visões de apoio, são uma característica comum nos principais SGBDs do mercado, chegando a compartilhar a mesma nomenclatura em alguns casos.

Separar o uso dessas tabelas da implementação principal do algoritmo, através da criação de visões, foi uma estratégia adotada para facilitar portabilidade do Lathe-DB, uma vez que características exclusivas do PostgreSQL ficam isoladas nessas visões.

Para facilitar a escrita dos algoritmos representamos os operadores da álgebra relacional na forma de função com a notação $\Re(f;T)$, onde $\Re$ representa um operador relacional qualquer e, os parâmetros, f e T representam respectivamente uma fórmula proposicional e uma relação. Por exemplo, dada uma seleção $\underset{id=1}{\sigma}(movies)$, representamos essa operação com a seguinte notação: $\sigma\big(id = 1; movies\big)$.

4.3.4.1 Algoritmo preProcessing

O Algoritmo *preProcessing*, recebe como entrada o nome do esquema do banco de dados, que se deseja realizar a busca por palavras-chave, e como saída, popula as tabelas que armazenam o Índice de Termos (IT), o Grafo do Esquema Relacional (GER) e o Dicionario de Atributos e Tabelas (DAT). O Algoritmo *preProcessing* internamente executa outros dois algoritmos, o *createSchemaGraph*, na Linha 1, e o *createTermIndex*, na Linha 2, e o funcionamento de ambos é detalhado a seguir.

Algoritmo 1: preProcessing

Entrada: [Nome do esquema do banco de dados S]
Saída : IT, GER e DAT materializados em tabelas

```
1 createSchemaGraph(S) ;
2 createTermIndex(attribute_dic) ;
```

4.3.4.2 Algoritmo createSchemaGraph

O Algoritmo *createSchemaGraph*, como o nome sugere, cria o Grafo do Esquema Relacional - GER e popula a tabela *attribute_dic*, ver Figura 11, que contém o Dicionário de Atributos e Tabelas - DAT.

Algoritmo 2: createSchemaGraph

Entrada: [Nome do esquema do banco de dados S]
Saída : GER e DAT materializados em tabelas

```
1 attribute_dic ← σ(table_schema = S; vw_sgbd_catalog);
2 foreach tuple ∈ vw_sgbd_schema_graph do
3 |   addEdge('schema_graph', source, destination, join_predicate);
4 end
```

Na Linha 1 são inseridos na tabela *attribute_dic* todas as tuplas da visão *vw_sgbd_catalog* que pertencem ao esquema fornecido como parâmetro de entrada, criando o Dicionário de Atributos e Tabelas - DAT, que será utilizado em seguida para montar o grafo do esquema relacional e posteriormente no processo de montagem das CNs. Na Linha 2 é criado um laço que percorre todas as tuplas da visão *vw_sgbd_schema_graph*. Na Linha 3, a função *addEdge* insere nas tabelas *vertice* e *edge*, ver Figura 12, a aresta correspondente as variáveis *source* e *destination* do grafo, nomeado nesse ponto de *schema_graph*, e esse laço se repete até que seja gerada a lista de adjacências que compõem o Grafo do Esquema Relacional - GER.

Para exemplificar como o GER é armazenado, tomamos novamente como exemplo o banco de dados do IMDB, ver Figura 15. Os dados persistidos nas tabelas *vertice* e *edge* são exemplificados na Figura 16.

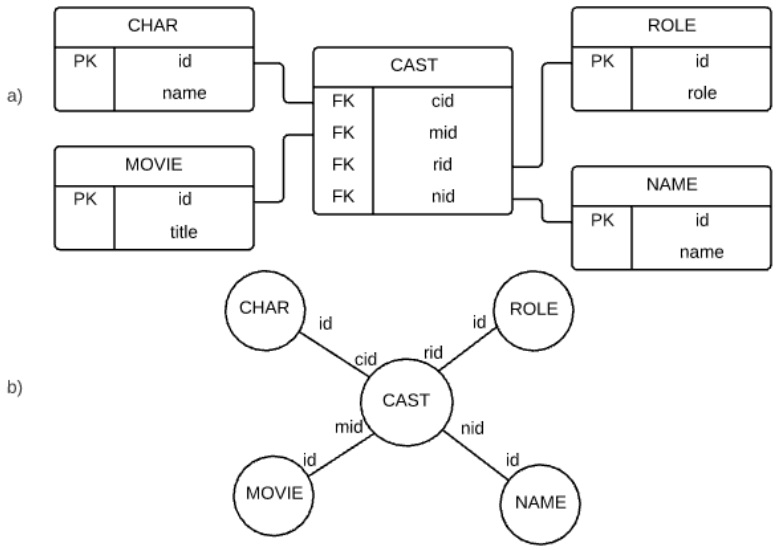


Figura 15 – a) Esquema relacional do banco de dados IMDB e b) sua representação em forma de grafo.

vertice	
graph_name	vertice
schema_graph	CHAR
schema_graph	MOVIE
schema_graph	ROLE
schema_graph	NAME
schema_graph	CAST

edges			
graph_name	source	destination	join_predicate
schema_graph	CHAR	CAST	CHAR.id =CAST.cid
schema_graph	MOVIE	CAST	MOVIE.id =CAST.mid
schema_graph	ROLE	CAST	ROLE.id =CAST.rid
schema_graph	NAME	CAST	NAME.id =CAST.nid

Figura 16 – Dados persistidos nas tabelas *vertice* e *edges*.

4.3.4.3 Algoritmo createTermIndex

O Algoritmo *createTermIndex*, ver Algoritmo 3, percorre o Dicionário de Atributos e Tabelas - DAT, e para cada atributo A, do tipo textual, monta uma instrução SQL que projeta todas as tuplas da tabela que contém o atributo A. Em seguida separa o termos distintos armazenados em cada uma dessas tuplas, alimentando uma tabela temporária, ver Figura 17, onde cada linha contém um termo individual e sua localização. Após esse processo, agrupa os termos na tabela *term_index* e para cada termo cria uma lista com as localizações do termo no banco de dados, criando assim o Índice de Termos, ver Figura 18.

Algoritmo 3: createTermIndex

Entrada: Tabela *attribute_dic*
Saída : Tabelas *temp_terms* e *term_index* populadas

```
1  temp_terms ← ∅ ;
2  foreach d ∈ (σ( column_type ∈ (char, varchar, text); attribute_dic)) do
3      foreach t ∈ (π(d.column_name; d.table_name)) do
4          term_list ← splitText (t.column_name);
5          foreach term ∈ term_list do
6              temp_terms ← temp_terms ∪ {(term, t.tid)};
7          end
8      end
9  end
10 term_index ← groupTerms(temp_terms) ;
11 fkj ← calculafkj(temp_terms);
12 calculaCk(fkj) ;
13 calculaNj(fkj) ;
```

temp_terms	
term	tuple
denzel	NAME.name,143
gangster	CHAR.name,452
washington	NAME.name,53
gangster	ROLE.role,4
gangster	MOVIE.title,83
denzel	NAME.name,19
washington	MOVIE.title,86
washington	NAME.name,12
america	MOVIE.tiltle,476

Figura 17 – Exemplo de dados persistidos na tabela temporária *temp_terms*.

Na Linha 1, é criada uma tabela temporária, em memória, onde onde cada linha armazena um termo e a sua localização, na forma $\langle termo, tabela.atributo.tid \rangle$. Na Linha

term_index	
term	tuple
denzel	{NAME.name,143,19}
washington	{NAME.name,143,12;MOVIE.title,86}
gangster	{CHAR.name,452; ROLE.role,4;MOVIE.title,83}
american	{MOVIE.tiltle,83}

Figura 18 – Exemplo de dados persistidos na tabela *term_index*.

de 2 é selecionada, a partir do DAT, uma lista de atributos do tipo texto (*char, varchar, text, etc*). Essa lista então é percorrida e na Linha 3, para cada atributo, é montada uma consulta SQL que seleciona todas as tuplas do banco de dados para aquele par atributo/tabela.

Na Linha 4, o texto contido no atributo de cada tupla é então dividido em uma lista de termos. Nessa etapa são ainda são aplicados alguns critérios de poda, como a remoção de *Stop words*[2]. E então, nas Linhas de 5 a 7, essa lista é percorrida, e cada termo armazenado na tabela temporária *temp_terms*. Na Linha 10, o conteúdo da tabela temporária *temp_terms* é agrupado em termos distintos, através da função *groupTerms* e as ocorrências desses termos na formam uma lista invertida, e então esse dados são persistidos na tabela *term_index*.

Finalmente, nas Linhas de 11 a 13, são calculadas as frequências dos termos, utilizadas na função de classificação das *Candidate Networks*, na Linha 11 a função $calculafkj$ soma quantidade de ocorrências de cada termo em relação a cada atributo do banco de dados e insere na Tabela fkj e na Linha 12, a função *calculaCk* soma quantidade de ocorrências de cada termo em relação ao banco de dados e insere na Tabela *ck*, já a Linha 13 soma a quantidade de termos que ocorrem em cada atributo e então atualiza a tabela *attribute_dic*, que abriga o Dicionário de Atributos e Tabelas.

4.3.4.4 Execução do Pré-Processamento

Por se tratar de uma etapa de pré processamento, todo o processamento do Algoritmo *preProcessing* precede a etapa de Geração das CNs, e seu tempo de execução varia conforme o tamanho do banco de dados em que se deseja realizar as consultas. Assumimos que essa etapa não será realizada com frequência, assim como ocorre em sistemas de busca. Para sistemas com grande volume de atualizações é possível, utilizado triggers, atualizar dinamicamente o Índice de Termos, porém essa questão será deixada para trabalhos futuros.

[2] *Stop words* são palavras consideradas irrelevantes para o conjuntos de resultados a ser exibido em uma busca por palavras-chave. Fonte: http://jmlr.org/papers/volume5/lewis04a/a11-smart-stop-list/english.stop

4.4 Geração das CNs

Nesta seção apresentamos o algoritmo utilizado na fase de Geração das *Candidates Networks*, chamado ***matCnGen***. Este algoritmo recebe como parâmetro de entrada uma lista de palavras-chave e, utilizando as estruturas criadas na fase de pré-processamento, gera as *Candidate Networks*. Nesse algoritmos são executados 4 passos: a Mineração de *Tuple-Sets*, Geração das *Query Matches*, Classificação das *Query Matches* e a Montagem das *Candidate Networks*, ver passos ①, ②, ③ e ④ da Figura 8, todos esses passos são explicados em detalhes mais adiante nesse capítulo.

Para facilitar o entendimento do funcionamento do Algoritmo *matCnGen*, a Figura 19 mostra a sua arquitetura de sua implementação e a sua interação com os algoritmos de apoio e as tabelas que compõem o Lathe-DB.

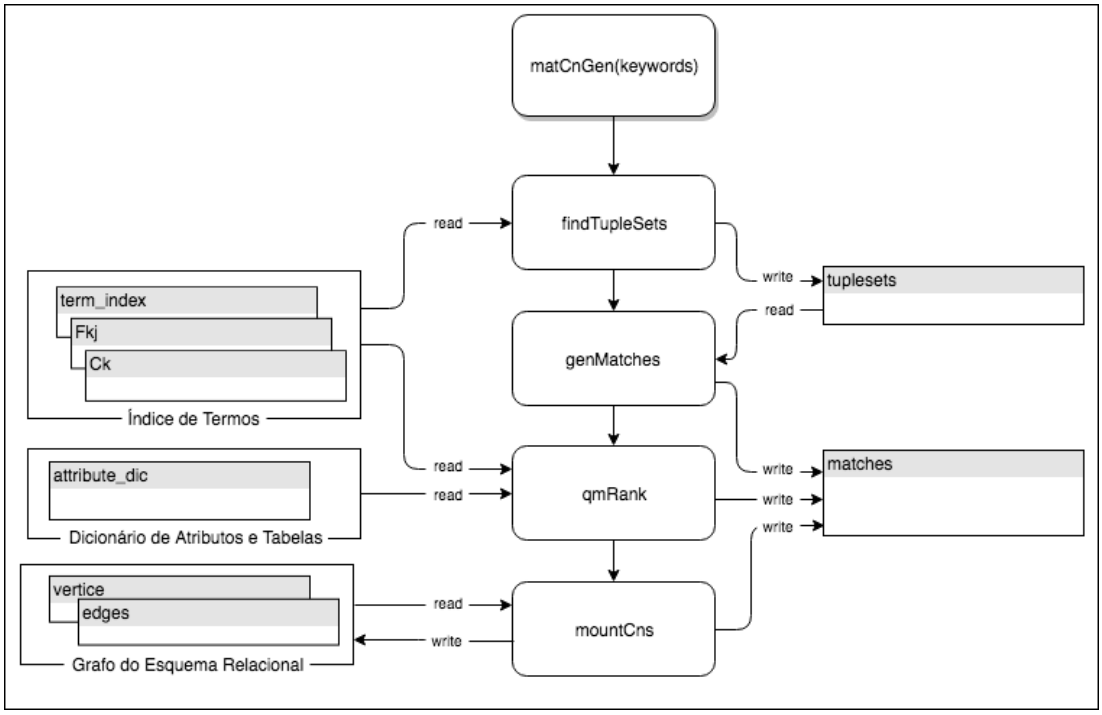


Figura 19 – Arquitetura do Algoritmo *matCnGen*.

4.4.1 Mineração de *Tuple-Sets*

A etapa de Mineração de *Tuple-Sets* é a primeira que ocorre na fase de Geração das CNs. Nessa etapa são computadas as interseções não-vazias das listas invertidas das ocorrências dos termos da consulta, armazenadas na tabela *term_index*, gerando assim os *termsets*, que são subconjuntos de termos da consulta, e suas co-ocorrências. Toda a operação de Mineração de *Tuple-Sets* é realizada na tabela *tuplesets*, ver Figura 20.

O algoritmo *findTupleSets* implementado é dividido em duas partes. Para exemplificar essa operação, vamos tomar como exemplo a instância da tabela *term_index* apresentada na Figura 10. Para a consulta $Q = \{denzel, washington, gangster\}$, vamos obter na primeira parte da execução do algoritmo, os *termsets* de tamanho 1, que são obtidos diretamente do Índice de Termos, e correspondem a cada palavra-chave da consulta submetida, ver Figura 21.

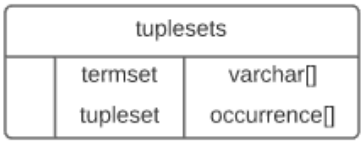


Figura 20 – Tabela que armazena os *tuple-sets*

Após isso, na segunda parte, outros *termsets*, que contém mais de uma palavra-chave, são construídos progressivamente computando as intersecções não-vazias das ocorrências nos *termsets* já encontrados. Como podemos observar na Figura 22, um novo *tupleset* foi criado na Linha 4. O algoritmo, ao identificar uma intersecção não-vazia entre os *tuplesets* $\{NAME.name, 143, 19\}$ da Linha 1 e $\{NAME.name, 143, 12; MOVIE.title, 86\}$ da Linha 2, onde a ocorrência $\{NAME.name, 143\}$ aparece em ambas, realizou as seguintes operações: Criou um novo registro na tabela *tupleset*, agrupando os *termsets* dos registros onde ocorreu a intersecção e um novo *tupleset* com a ocorrência, e removeu a ocorrência que gerou a intersecção dos *tuplesets* originais.

tuplesets		
#	termset	tupleset
1	{denzel}	{NAME.name,143,19}
2	{washington}	{NAME.name,143,12;MOVIE.title,86}
3	{gangster}	{CHAR.name,452; ROLE.role,4;MOVIE.title,83}

Figura 21 – Primeira parte da execução do algoritmo *findTupleSets*.

tuplesets		
#	termset	tupleset
1	{denzel}	{NAME.name,19}
2	{washington}	{NAME.name,12;MOVIE.title,86}
3	{gangster}	{CHAR.name,452; ROLE.role,4;MOVIE.title,83}
4	{denzel,washington}	{NAME.name,143}

Figura 22 – Segunda parte da exeução do algoritmo *findTupleSets*.

Assim como no Lathe [Oliveira e Silva 2015, Oliveira e Silva 2017], em nosso trabalho nós mapeamos o problema de encontrar os termos da consulta que ocorrem em uma mesma tupla como o problema de mineração de itens frequentes. Que é um problema da área de Mineração de Dados, com vários algoritmos bem conhecidos, como o Apriori [Agrawal, Imielinski e Swami 1993], FP-grow [Han, Pei e Yin 2000] , ECLAT [Zaki 2000] e etc. O algoritmo *findTupleSets* é baseado no algoritmo ECLAT (ver Seção 3.1.1).

4.4.2 Geração das *Query Matches*

O Algoritmo de Geração das *Query Matches*, chamado *genMatches*, gera as *Query Matches* de acordo com os conceitos apresentados no Lema 1, ver subseção 3.1.2. A estratégia adotada para gerar os *Query Macthes* consiste em combinar todos os *tuple-sets* obtidos no Algoritmo 5, que foram persistidos na tabela *tupleset*, ver Figura 20, de forma que todos os *set coverings* gerados contenham todos os termos da consulta.

Como entrada o algoritmo recebe uma lista de palavras-chave, e como saída o algoritmo popula a tabela *matches* (Figura 23).

matches	
tupleset	occurrence[]
cover	varchar[]
joins	text
predicates	text
cn	text
rank	double

Figura 23 – Tabela que armazena as *Query Matches*

Por exemplo, considere a consulta $Q = \{charlie, sheen\}$, e suponha que foram gerados os *tuple-sets* da Figura 24.

tuplesets		
#	termset	tupleset
1	{charlie}	{CHAR.name,32}
2	{charlie, sheen}	{NAME.name,2}

Figura 24 – Tabela *tupleset* após a execução do algoritmo *findTupleSets* para a consulta $Q = \{charlie, sheen\}$.

Primeiramente o algoritmo adiciona o *placeholder* # em cada *tuple-set*, ficando com:

$T1 = \{CHAR.name, 32; \#\}$ com *set covering* $\{charlie\}$ e

$T2 = \{NAME.name, 2; \#\}$ com *set covering* $\{charlie, sheen\}$

O *placeholder* # adicionado é necessário para maximizar as combinações, que são feita através do produto cartesiano $T1 \times T2$, que gera as seguintes combinações:

$m_1 = \{NAME.name, 2; CHAR.name, 32\}$ com *set covering* $\{charlie, sheen; charlie\}$,

$m_2 = \{NAME.name, 2; \#\}$ com *set covering* $\{charlie, sheen; \#\}$,

$m_3 = \{\#; CHAR.name, 32\}$ com *set covering* $\{\#; charlie\}$ e

$m_4 = \{\#; \#\}$ com *set covering* $\{\#; \#\}$

A seguir são removidos os *placeholders*, ficando:

$m_1 = \{NAME.name, 2; CHAR.name, 32\}$ com *set covering* $\{charlie, sheen; charlie\}$,

$m_2 = \{NAME.name, 2\}$ com *set covering* $\{charlie, sheen\}$,

$m_3 = \{CHAR.name, 32\}$ com *set covering* $\{charlie\}$ e

$m_4 = \{\}$ com *set covering* $\{\}$

Logo após são descartadas todas as combinações onde o *set covering* não contém todas as palavras-chave da consulta Q, resultado nos seguintes *matches*:

$m_1 = \{NAME.name, 2; CHAR.name, 32\}$ com *set covering* $\{charlie, sheen; charlie\}$ e

$m_2 = \{NAME.name, 2\}$ com *set covering* $\{charlie, sheen\}$

As *Query Matches* geradas nesse ponto são persistidas na tabela *matches*, ver Figura 25, que será gradualmente preenchida nos próximos passos do Algoritmo *matCnGen*.

matches					
tupleset	cover	joins	predicates	cn	rank
{NAME.name,2 ;CHAR.name,32,}	{charlie, sheen; charlie}				
{NAME.name,2}	{charlie; sheen}				

Figura 25 – Tabela *matches* após a execução do algoritmo *genMatches*.

4.4.3 Classificação de Query Matches

Na etapa de Classificação de *Query Matches*, passo ③ da Figura 8, foi utilizado como base algoritmo cnRank [Oliveira, Silva e Moura 2015], o mesmo utilizado no Lathe para classificar CNs. Porém, diferente do Lathe onde as CNs são previamente montadas e em seguida classificadas, adotamos a estratégia de classificar as *Query Matches* antes de montar as CNs, dessa forma otimizamos o processo montando as

CNs somente a partir das *Query Matches* melhores classificadas. Dessa forma, ao atingir as *top-k CNs*, o processo de montagem é interrompido não necessitando que todas as CNs sejam montadas, economizando assim tempo de processamento.

O algoritmo de Classificação de *Query Matches*, chamado *qmRank*, atribui um peso para cada *Query Match* a partir dos valores que foram pré-calculados durante a criação do *Índice de Termos*, aplicando o conceito de *Term Frequency (TF)* e *Inverse Attribute Frequency (IAF)*, que foi descrito em detalhes na seção 3.1.5, utilizando a fórmula:

$$w_{fk} = tf \times iaf = \frac{log(1 + Fkj)}{log(1 + Nj)} \times log\left(1 + \frac{Na}{Ck}\right)$$

Onde *Fkj* corresponde ao número de ocorrências de cada termo em cada atributo, que foi calculado e persistido na tabela de mesmo nome, ver Figura 10. Nj é número total de termos distintos que ocorrem em cada atributo textual, que é obtido através do atributo *Nj* da tabela *attribute_dic*, ver Figura 11.

Para se obter o valor *Na*, que é o número total de atributos no na banco de dados, são contadas as tuplas da tabela *attribute_dic*. E por fim *Ck*, é o número total de atributos onde cada termo ocorre, obtido diretamente da tabela *Ck*.

O algoritmo *qmRank*, percorre todas as tuplas da tabela *matches*, calculando o peso de cada *Query Match*. Seguindo com nosso exemplo, ao final do processamento do algoritmo atualiza a tabela *matches* como podemos observar na Figura 26.

matches					
tupleset	cover	joins	predicates	cn	rank
{NAME.name,2 ;CHAR.name,32,}	{charlie, sheen; charlie}				1.017964157
{NAME.name,2}	{charlie; sheen}				0.036833588

Figura 26 – Tabela *matches* após a execução do algoritmo *qmRank*.

4.4.4 Montagem e Seleção de CNs

Conforme descrito na Seção 3.1.4, sabemos que as *Query Macthes* contém todos os *non-free tuple-sets* porém, para montar uma CN, é necessário localizar os *free tuple-sets* que os conectam e formam uma árvore que une os termos da consulta. Neste ponto o algoritmo *mountCns* verifica se e possível conectar os *non-free tuple-set* e os *free tuple-sets* percorrendo o grafo do esquema relacional do banco de dados. Tomando como exemplo a Figura 26, a *Query Match* da linha 1, é representada na Figura 27.

$$\sigma_{(name \supseteq \{charlie, sheen\};NAME)}, \sigma_{(name \supseteq \{charlie\};CHAR)}$$

Figura 27 – Query Match.

Dessa forma, para ser montada a CN, o algoritmo deve localizar o(s) *free tuple-set(s)* que conecta(m) as tabelas NAME e CHAR. No grafo utilizado na Figura 28 podemos observar que para ligar as duas tabelas é necessário utilizar a tabela CAST, formando um novo grafo que representa a *Candidate Network*

Assim como proposto no Lathe, o problema de ligar as tabelas envolvidas em um *match*, foi mapeado para o problema das *Steiner Trees*, onde dado um conjunto de pontos denominados terminais, os *non-free tuple-sets*, existe um grafo mínimo que os conecta interligados pelos vértices chamados *Steiner Nodes*, em nosso caso os *free tuple-sets*. Da mesma forma que o Lathe, o algoritmo *mountCns* é baseado no algoritmo de Takahashi [Takahashi e Matsuyama 1980].

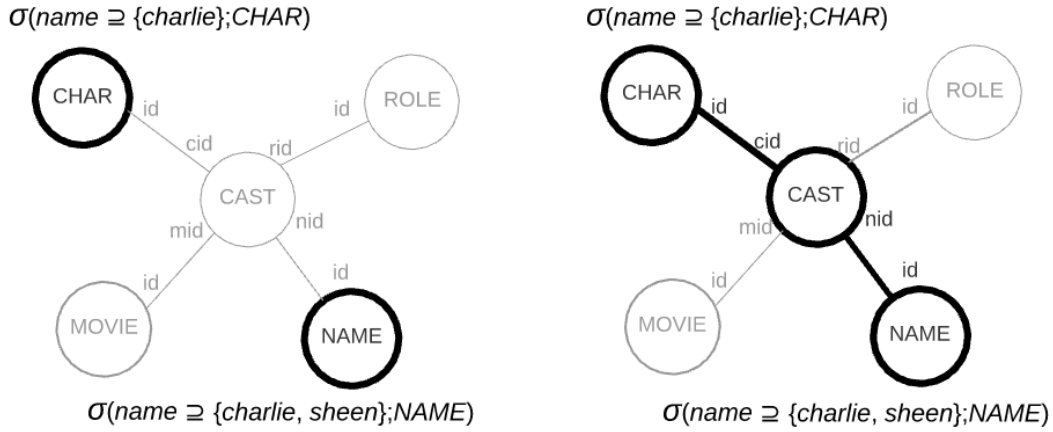


Figura 28 – Duas etapas da criação de um novo grafo a partir da ligação dos *non-free tuple-sets* e os *free tuple-sets*.

Com a junção dos *non-free tuple-sets* e o *free tuple-set*, nossa *Candidate Network* é representada na Figura 29. Caso não seja possível realizar a junção dos *free tuple-sets* percorrendo o Grafo do Esquema Relacional, o *match* é descartado.

$$\sigma_{(name \supseteq \{charlie, sheen\};\ NAME)} \bowtie \sigma_{(NAME.id = nid \wedge CHAR.id = cid;\ CAST)} \bowtie \sigma_{(name \supseteq \{charlie\};CHAR)}$$

Figura 29 – *Candidate Network*

Após criar o grafo que representa a *Candidate Network*, o algoritmo *mountCns* atualiza gradativamente a tabela *matches*, em um primeiro passo atualiza o atributo

joins, preenchendo o mesmo parte do código SQL que faz a junção entre as tabelas *NAME*, *CAST* e *CHAR*. Em seguida atualiza o atributo *predicates*, também com parte do código SQL que seleciona as tuplas que contém as palavras-chave da consulta do usuário. E finalmente preenche o atributo *cn* com o código SQL correspondente a *Candidate Network*, nesse ponto o estado da tabela *matches* pode ser observado na Figura 30.

matches					
tupleset	cover	joins	predicates	cn	rank
{NAME.name,2 ;CHAR.name,32,}	{charlie, sheen; charlie}	NAME.id=CAST.nid and CHAR.id=CAST.cid	NAME.name ilike '%charlie% and NAME.name ilike '%sheen% and CHAR.name ilike '%charlie%'	select * from NAME, CAST, CHAR where NAME.id=CAST.nid and CHAR.id=CAST.cid and NAME.name ilike '%charlie% and NAME.name ilike '%sheen% and CHAR.name ilike '%charlie%'	1.017964157
{NAME.name,2}	{charlie; sheen}	NAME	NAME.name ilike '%charlie% and NAME.name ilike '%sheen%	select * from NAME where NAME.name ilike '%charlie% and NAME.name ilike '%sheen%	0.036833588

Figura 30 – Tabela *matches* após a execução do algoritmo *mountCns*.

4.4.5 Algoritmo *matCnGen*

O Algoritmo *matCnGen*, ver Algoritmo 4, recebe como entrada a lista de palavras-chave da consulta Q e retorna uma lista C de *Candidate Networks* já classificadas para o usuário. Opcionalmente também podem ser informados como entrada os parâmetros, *topK* e *Tmax*, onde *topK* indica que serão geradas somente as k *CandidateNetworks* com maior peso e *Tmax* que limita o número máximo de vértices que podem existir no sub-grafo gerado na fase de Montagem e Seleção das *Candidate Networks*, o uso desses parâmetros é detalhado na Seção 4.5.4.

Nas Linhas de 1 a 4 são executados respectivamente os algoritmos *findTupleSets*, *genMatches*, *qmRank* e *mountCns*, que são detalhados mais adiante nessa seção, e na Linha 5 são retornadas todas as *Candidates Networks* geradas e classificadas de acordo com seu peso.

4.4.5.1 Algoritimo findTupleSets

O Algoritmo *findTupleSets* recebe como parâmetro de entrada uma lista Q com as palavras-chave da consulta submetida pelo usuário, e como saída o algoritmo popula

Algoritmo 4: matCnGen

Entrada: *Keyword Query* $Q = \{k_1, k_2, \ldots, K_m\}$, *[topK, Tmax]*
Saída : *Candidate Networks* $C = \{cn_1, cn_2, \ldots, cn_n\}$

```
1 findTupleSets(Q);
2 genMathes(tuplesets, Q);
3 qmRank(macthes) ;
4 mountCns(macthes, topK, Tmax) ;
5 C ← π(cn;(τ(rank desc; matches)))
```

Algoritmo 5: findTupleSets

Entrada: *Keyword Query* $Q = \{k_1, k_2, \ldots, k_m\}$
Saída : A tabela *tupleset* populada

```
1  tupleset ← ∅;
2  foreach ts ∈ (σ(term ∈ Q ; term_index)) do
3  |  tupleset ← tupleset ∪ {(ts.term, ts.occurences)};
4  end
5  have_intersection ← true;
6  while have_intersection do
7  |  have_intersection ← false;
8  |  foreach pts ∈ (τ(rowid asc; tuplesets)) do
9  |  |  foreach ots ∈ (τ(rowid asc; (σ( id > pts.id ; tuplesets)))) do
10 |  |  |  Intersection ← pts.tupleset ∩ ots.tupleset;
11 |  |  |  if Intersection ≠ ∅ then
12 |  |  |  |  have_intersection ← true;
13 |  |  |  |  termset_union ← pts.termset ∪ ots.termset;
14 |  |  |  |  tuplesets ← tuplesets ∪ {(termset_union, Intersection)});
15 |  |  |  |  removeIntersection (ots.id, pts.id, Intersection);
16 |  |  |  end
17 |  |  end
18 |  end
19 end
20 tuplesets ← tupleset−σ( tupleset = ∅; tupleset);
```

a tabela *tuplesets*, ver Figura 20.

Na Linha 1, são apagados todas as tuplas da tabela *tupleset*, ficando preparada para receber os novos *tuple-sets*. Nas Linhas de 2 a 4, é realizada uma consulta na tabela *term_index*, onde são retornadas todas as listas invertidas dos termos que pertencem a lista Q de palavras-chave, em seguida, na Linha 3 cada uma das tuplas retornadas e inserida na tabela *tupleset*. Dessa forma são gerados os primeiros *tuple-sets*, que tem correspondência direta com o Índice de Termos, ou seja, nesse ponto são criados os *tuples sets* cujos *termsets* contém cada um dos termos da consulta, formando *termsets* de tamanho 1.

Na Linha 5, e inicializada uma variável de controle, chamada *have_intersection*, que ira fazer com que sejam realizadas sucessivas iterações, que compreendem as Linhas de 6 a 19, sobre o conjunto de *tuple-sets* gerados até que não ocorra nenhuma nova interseção. Na Linha 7, é atribuído o valor *false* a variável de controle *have_intersection*, no inicio de cada iteração para garantir que não ocorra nenhuma nova iteração caso não ocorra uma nova interseção de *tuplesets*.

Na Linha 8, é inicializado um laço *pts* tendo como base uma consulta sobre a tabela *tupleset*. Vale ressaltar aqui que essa é uma consulta ordenada pelo atributo *rowid*, que é do tipo inteiro que identifica cada tupla. Na Linha 9 é inicializado um laço *ots* que também consulta a tabela *tuplesets*, porém, além de ordenar pelo atributo *id* só são selecionados as tuplas com o *id* maior que *pts.id*, ou seja essa combinação de laços faz com que cada par de tuplas da tabela *tuplesets* sejam processadas de forma ordenada.

Na Linha 10 é realizada a operação de intersecção entre *pts.tupleset* e *ots.tupleset* e o conjunto resultante é armazenado na variável *intersection*, e na Linha 11 é verificado se o resultado na operação não é um conjunto vazio. Caso o conjunto *intersection* não seja vazio, na Linha 12 a variável de controle *have_intersection* é atualizada garantindo novamente que uma nova iteração aconteça e todas as tuplas sejam novamente processadas. Na Linha 13 é armazenado em *termset_union* a união dos termos contidos em *pts.termset* e *ots.termset*. Na Linha 14 é criada uma nova tupla na tabela *tupleset* com os valores contidos em *have_intersection* e *intersection*, e na Linha 15 a função *removeIntersection* remove do conjunto de *tuplesets* das tuplas identificadas pelos ids *pts.id* e *pts.id* a intersecção armazenada na variável *intersection*. Finalmente na Linha 20 são removidas da tabela *tuplesets* as tuplas onde o atributo *tupleset* ficou vazio após o processamento das sucessivas remoções das interseções realizadas pela função *removeIntersection* na Linha 15. Toda essa operação cria os *termsets* de tamanho maior que 1, como podemos observar no exemplo da Linha 4, na Figura 22.

4.4.5.2 Algoritmo genMatches

Na Linha 1, é criada uma tabela temporária R, que é populada através de uma projeção na tabela *tupleset*. Durante a projeção é adicionado aos conjuntos, contidos nos atributos *termset* e *tupleset* o *placeholder* #, para maximizar as combinações.

Na Linha 2, cada conjunto de *tuple-sets* e termsets em R é combinado por produto cartesiano e é armazenada em uma tabela temporária M.

Nas Linhas de 3 a 9, a tabela temporária M é percorrida e são realizadas duas operações. Nas Linhas 4 e 5 , o *placeholder* # é removido, e na Linha 6 é verificado se cada *set covering*, que foi gerado pelo produto cartesiano, contém todos os termos da consulta e se após a remoção do *placeholder* # o conjunto ficou vazio, se qualquer uma dessas verificações for verdadeira a tupla é removida. E, finalmente na Linha 10 a tabela temporária M é persistida na tabela *matches*.

Algoritmo 6: genMatches

Entrada: A tabela *tuplesets* populada, *Keyword Query* $Q = \{k_1, k_2, \ldots, k_m\}$
Saída : A tabela *matches* populada

```
1  R ← π(tupleset ∪{#}, termset ∪ {#}; tuplesets) ;
2  M ← R1 × ... × Rn;
3  foreach m ∈ M do
4  |   m.termset ← m.termset − {#};
5  |   m.tupleset ← m.tupleset − {#};
6  |   if (m.termset ⊅ Q) ∨ (m.termset = ∅) then
7  |   |   M ← M − m;
8  |   end
9  end
10 matches ←ρ(tupleset, cover; M);
```

4.4.5.3 Algoritmo qmRank

Os algoritmos *qmRank* e *getWeight*, descritos a seguir são a aplicação direta do modelo TF-IAF [Mesquita et al. 2007]. .

Algoritmo 7: qmRank

Entrada: A tabela *matches* populada
Saída : A tabela *matches* classificada

```
1 M ← π(tupleset,cover,rank ; matches);
2 Na ← γ(count(∗); attribute_dic);
3 foreach m ∈ M do
4 |   m.rank ← getWeight(m.tupleset, m.cover, Na);
5 end
6 matches ← M;
```

Na Linha 1 é criada a tabela temporária M que é populada por uma projeção na tabela *matches* e na Linha 2 é atribuído o valor de Na, que é o número total de atributos no banco de dados, que será utilizado no cálculo do peso da *Query Match*. Nas Linhas de 3 a 5 a tabela temporária M é percorrida e, na Linha 4, é atribuído um peso para cada *macth*, através da função *getRank*. Por fim a tabela *matches* é atualizada.

A função *getRank* recebe como parâmetros, um *array* de *tuplesets*, um array de *covers* e o valor de Na. Para cada item no *array* de *tuplesets* existe um item correspondente no *array* de *covers*, dessa forma cada *tupleset* tem seu conjunto de termos que formam o *set covering* da consulta do usuário.

Algoritmo 8: getWeight

Entrada: $tupleset, cover, Na$
Saída : $weight$

1 $weight \leftarrow 0$;
2 **for** i *to* $size(tupleset)$ **do**
3 $\quad weight \leftarrow size(cover[i]) - 1$;
4 $\quad$ **foreach** $t \in cover[i]$ **do**
5 $\quad\quad Fkj \leftarrow \pi(\ fkj\ ;\ (\sigma(term = t; Fkj)))$;
6 $\quad\quad Ck \leftarrow \pi(\ ck\ ;\ (\sigma(\ term = t\ ; \mathrm{Ck})))$;
7 $\quad\quad Nj \leftarrow \pi(\ nj\ ;\ (\sigma(\ column\_name{=}tupleset[i].attribute\ ;\ \mathrm{attribute\_dic})))$
8 $\quad\quad weight \leftarrow weight + \frac{log(1+Fkj)}{log(1+Nj)} \times log\left(1 + \frac{Na}{Ck}\right)$;
9 $\quad$ **end**
10 **end**
11 $weight \leftarrow weight \times (1/size(tupleset)$

No algoritmo *getWeight*, ver Algoritmo 8, na Linha 1 a variável *rank* é inicializada, e na Linha 2 é iniciado um laço que realiza n repetições, de acordo com o número de *tuplesets* informados como entrada. Já na Linha 3 a variável *rank* é incrementada com a quantidade de termos contidos em cada elemento do *array* de *covers* menos 1, ou seja caso exista mais de um termo no elemento do *array* de *covers* a atribuímos um peso maior para o *match* porque isso indica que os termos na consulta ocorrem juntos em uma tupla.

Na Linha 4 o *array* de *covers* é percorrido, e nas Linhas de 5 a 7 os valores das variáveis *Fki*, *Ck* e *Nj* são atribuídos com os valores persistidos nas tabelas que compõem o *Indice de Termos*, ver Figura 10. Na Linha 8 é incrementada a variável *weight* aplicando o modelo TF-IAF, descrito na seção 3.1.5. Por fim, na Linha 11 são penalizados os *matches* que contêm muitos *tuplesets*, porque *matches* que contêm muitos *tuplesets* indicam que os termos da consulta ocorrem "distantes"uns dos outros o que indica possuem menor relevância para o usuário [Goldman et al. 1998].

4.4.5.4 Algoritmo mountCNs

O Algoritmo *mountCns*, ver Algoritmo 9, recebe como entrada a tabela *matches*, já classificada pelo Algoritmo *qmRank*, e a atualiza montando as *Candidate Networks*, utilizando as informações já salvas na tabela *matches*.

Algoritmo 9: mountCns

```
   Entrada: A tabela matches classificada, topK, Tmax
   Saída  : A tabela matches atualizada
1  G_TS ← getGER();
2  M ←(τ( desc(rank) ; matches));
3  numCNs ← 0;
4  foreach m ∈ M do
5  |  m.joins ← getMinimalSteinerTree(G_TS, m.tupleset, Tmax);
6  |  if m.joins ≠ ∅ then
7  |  |  m.predicates ← getPedicates(m.tuplesets, m.cover);
8  |  |  m.cs ← mountCn(m.joins, m.predicates);
9  |  |  numCNs ← numCNs + 1;
10 |  end
11 |  if numCNs = topK then
12 |  |  exit;
13 |  end
14 end
15 matches ← σ(joins ≠ ∅; M);
```

Na Linha 1 o Grafo do Esquema Relacional - GER, que foi previamente criado na etapa de pré-processamento, é armazenado em G_{TS}. Na Linha 2 é armazenado na tabela temporária M as tuplas da tabela *matches*, classificadas ordem decrescente pelo atributo *rank*, de forma que, ao realizar a iteração sobre essa tabela temporária, as *Query Matches* de maior peso serão processadas primeiro.

Na Linha 3, é inicializada a variável *numCNs*, essa é a variável que pode, caso o número de *matches* seja muito grande, interromper a montagem de *Candidate Networks* quando o número de CNs atingir o topK, ou seja assim que o algoritmo conseguir montar a quantidade de CNs correspondente a variável topK, as iterações sobre a tabela *matches* param, otimizando assim o tempo de processamento do algoritmo.

Na Linha 4 essa tabela temporária M é percorrida e na Linha 5, para cada *match* m é chamada a função *getMinimalSteinerTree*. Essa função recebe como parâmetros o Grafo do Esquema Relacional, o *tupleset* e o *cover* da tupla corrente da tabela M. Caso a função consiga conectar as tabelas que constam no *tupleset* a função retorna uma *string* que será utilizada para montar as junções da CN, caso contrário retornará nulo, indicando que é impossível conectar os *nom-free tuple-sets* que formam o *match*.

Na Linha 6, caso o atributo *joins* esteja populado, a função *getPredicates*, na Linha

7, retorna os predicados utilizados para compor a CN. Os predicados são a parte da CNs onde serão filtradas as tuplas que contém as palavras-chave, ou seja as *nom-free tuple-sets*. Em seguida na Linha 8 a função *mountCn* cria a instrução SQL que representa a CN, utilizando as *strings* já armazenadas nos atributos *joins* e *predicates*.

Na Linha 11 o algoritmo verifica se o número de CNs montadas até o momento atingiu o número informado no parâmetro *topK*, se sim interrompe as interações. E finalmente na Linha 15, a tabela *matches* é atualizada, são persistidos os dados da tabela temporária *M*, podando os *matches* onde não foi possível conectar os *nom-free tuple-sets*.

4.5 Funcionamento do Lathe-DB

O funcionamento do Lathe-DB é dividido em três etapas: a de Preparação, Pré-processamento e a de Execução. A etapa de Preparação, que é executada somente uma vez, cria a estrutura do Lathe-DB. O Pré-processamento é executado no banco de dados pelo menos uma vez, ou quando ocorrerem atualizações, deixando-o pronto para a busca por palavras-chave. A etapa de Execução ocorre sempre que uma busca por palavras-chave é realizada.

4.5.1 Preparação

A etapa de preparação, como o nome sugere, prepara o banco de dados criando, em um esquema separado chamado *lathedb*, todos os objetos necessários a operação do Lathe-DB. Ao todo são criadas 11 tabelas, 31 funções e 2 visões.

A decisão de criar um esquema separado para abrigar os objetos utilizados pelo Lathe-DB tem dois motivos práticos. O primeiro é de isolar os objetos do Lathe-DB dos demais objetos do banco de dados, visando uma maior organização. Segundo, ao criar um esquema separado, os administradores de banco de dados podem optar por criar esse esquema em uma estrutura de armazenamento independente da utilizada no banco de dados principal, otimizando assim as operações leitura e escrita em disco.

4.5.2 Pré-processamento

Conforme já explicado na Seção 4.3, a etapa de pré-processamento realiza três operações distintas. Cria o Dicionário de Atributos Tabelas, o Grafo do Esquema Relacional e o Índice de Termos. Para executar o pré-processamento basta executar a função *preProcessing()*, lembrando que a função *preProcessing* pode, opcionalmente, receber como parâmetro o nome do esquema que agrupa as tabelas. Caso não seja informado o esquema, a função *preProcessing()* vai processar por padrão o esquema

public. No exemplo do Código 1, podemos observar a função *preProcessing* em ação que, ao ser executada, exibe mensagens de *feedback* para o usuário informando os passos que estão sendo executados.

```
imdb_subset=# SELECT * FROM lathedb.preProcessing();
NOTICE: 21:40:43.83215 Creating Attribute Dictionary ...
NOTICE: 21:40:43.86373 Creating Schema Graph ...
NOTICE: 21:40:43.87285 Creating Term Index ...
NOTICE: 21:47:13.44879 Calculating fkj ...
NOTICE: 21:48:42.77277 Calculating Nj ...
NOTICE: 21:48:42.90512 Calculating Ck ...
NOTICE: Time to index... 00:08:01.211937
NOTICE: 271286 term(s) indexed
 preprocessing
---------------

(1 row)

imdb_subset=#
```

Código 1 – Exemplo da execução do Pré-Processamento.

4.5.3 Execução

Na etapa de execução, o Lathe-DB está pronto para gerar as *Candidates Networks*, e como podemos observar na Listagem 2, o uso da função *matCnGen()* é bem simples. No exemplo foi informado o único parâmetro obrigatório, que é um texto, delimitado por aspas simples que contém uma lista de palavras-chave separadas por espaços em branco.

```
imdb_subset=# SELECT * FROM latheDB.matCnGen('Denzel Washington Gangster');
NOTICE: Creating TupleSets ...
NOTICE: 4 tupleset(s) found.
NOTICE: Generating Query Matches ...
NOTICE: 440 Matches created.
NOTICE: Ranking Query Matches...
NOTICE: Assembling Candidate Networks ...
```

Código 2 – Exemplo de uso da função *matCnGen()*.

Ao executar a função *matCnGen*, da mesma forma que a função *preProcessing()*, são geradas mensagens de *feedback* para o usuário exibindo os passos que são realizados. Ao final desses passos a função retorna uma lista de *Candidate Networks* para as palavras-chave informadas como parâmetro, como podemos observar no Código 3[3] .

[3] Quebras de linha foram inseridas no exemplo para facilitar a visualização.

```
                                                candidate_network
-------------------------------------------------------------------------------------------------------------
SELECT *
FROM name,title,cast_info
WHERE cast_info.personismo=name.id
AND cast_info.movie_id=title.id
AND to_tsvector('english',name.name) @@ to_tsquery('english','denzel')
AND to_tsvector('english',name.name) @@ to_tsquery('english','denzel,washington')
AND to_tsvector('english',name.name) @@ to_tsquery('english','washington')
AND to_tsvector('english',title.title) @@ to_tsquery('english','gangster');

 SELECT * FROM name,title,cast_info
 WHERE cast_info.person_id=name.id
 AND cast_info.movie_id=title.id
 AND  to_tsvector('english',name.name) @@ to_tsquery('english','denzel,washington')
 AND to_tsvector('english',title.title) @@ to_tsquery('english','gangster');
(2 rows)
```

Código 3 – *Candidate Networks* geradas pela função *matCnGen()*.

A execução das *Candidates Network* fica sobre responsabilidade do desenvolvedor, ou administrador do banco de dados, que vai escolher a forma mais conveniente de apresentar os dados.

4.5.4 Sintaxe

Conforme foi comentando na sessão 4.4.5, estão disponíveis outros parâmetros opcionais, ver Código 4, que alteram o comportamento da função *matCnGen*, e são detalhados a seguir:

```
SELECT * FROM latheDB.matCnGen ( keywords [, Tmax, topK, fType, onlyIndexAtt]);
```

Código 4 – Parâmetros da função *matCnGen*.

- ***keywords***: Informa para a função *matCnGen* quais são as palavras-chave da consulta do usuário. O parâmetro é do tipo texto e os termos da consulta devem ser separados por espaços em branco e é o único obrigatório.

- ***Tmax***: O parâmetro *Tmax* limita o número máximo de vértices que podem existir nas *Steiner Trees* geradas pela função *getSteinerMinimalTree*, ver Algoritmo 9. O parâmetro é do tipo numérico e caso não seja informado o valor default para *Tmax* é 10.

- ***topK***: O parâmetro *topK* indica que serão processados uma quantidade X de *Query Matches* até atingir o valor informado no parâmetro *topK*. O parâmetro é do tipo numérico e caso não seja informado o valor default para *topK* é 10.

- ***sType***: O parâmetro *sType* pode receber os valores 1 para utilizar o operador *ilike* ou 2 para utilizar a combinação de funções *to_tsvector* e *to_tsquery*, conforme observado no exemplo do Código 3. O operador *ilike*, ver Código 5, é utilizado para localizar uma sequencia de caracteres contida em um texto não diferenciando maiúsculas de mínusculas. Já a combinação de funções *to_tsvector* e *to_tsquery* utiliza as funcionalidades de busca textual do PostgreSQL, foram providas essas duas formas porque, durante os experimentos, foi identificado que o operador *ilike* tem uma performance melhor na busca onde os atributos do tipo *text*, que tem o conteúdo muito grande (com alguns *megabytes*). O parâmetro é do tipo numérico e caso não seja informado o valor default para *sType* é 2.

```
SELECT *
FROM name,title,cast_info
WHERE cast_info.personismo=name.id
AND cast_info.movie_id=title.id
AND name.name ilike '%denzel%'
AND name.name ilike '%washington%'
AND title.title ilike '%gangster%';
```

Código 5 – Exemplo de CN utilizando o operador ilike.

- ***onlyIdAtt***: O parametro *onlyIndexAtt* pode receber os valores *true*, para montar a CN utilizando somente os atributos que foram indexados e *false* para montar a CN utilizando o * (asterisco) para que sejam retornados todos os atributos de todas as tabelas que compõe a *Candidate Network*.

5 Avaliação Experimental

Conforme já comentado, o objetivo principal deste trabalho é integrar um mecanismo de busca por palavras-chave em um SGBD, implementando o método Lathe [Oliveira e Silva 2015, Oliveira e Silva 2017] no ambiente do PostgreSQL. Dessa forma nossa avaliação experimental consiste em comparar os resultados obtidos no Lathe-DB com os resultados do Lathe, demonstrando assim sua viabilidade.

5.1 Configuração dos Experimentos

Nesta sessão, apresentamos a configuração utilizada para execução dos experimentos para comparação dos resultados entre o Lathe e o Lathe-DB. Os resultados dos experimentos são apresentados nas sessões subsequentes.

Note que, por estarmos utilizando a mesma configuração que o Lathe para nossos experimentos, com exceção do *hardware*, os números apresentados, referentes aos *datasets*, *query sets*, gabaritos e demais artefatos utilizados, são também os mesmos do Lathe, o que valida nosso objetivo principal de demonstrar a viabilidade a aplicação do método proposto no Lathe no ambiente do SGBD.

5.1.1 *Hardware*

Nossos experimentos foram executados em um máquina com processador Intel Core i7 3 GHz (8 GB RAM, 500GB HD) executando o sistema operacional Ubuntu Linux 14.04 LTS. Usamos uma instalação padrão do PostgreSQL, versão 9.4, em sua configuração padrão, sem a adição de módulos adicionais que estendessem sua funcionalidade ou performance.

5.1.2 *Datasets*

Para fins de comparação, utilizamos os mesmos conjunto de dados [1], ou *datasets*, utilizados pelo Lathe, e também por muitos outros sistemas R-KwS [Coffman e Weaver 2010, Luo et al. 2007, Oliveira, Silva e Moura 2015, Coffman e Weaver 2010, INEX 2011]. Foram utilizados cinco *datasets*: Mondial, IMDB, Wikipedia, DBLP e TPC-H. Na Tabela 1, são apresentados alguns detalhes desses *datasets*, incluindo seu tamanho em *megabytes*, número de tabelas, a quantidade total de tuplas e o número de Restrições de Integridade Referencial (RIR) em seus esquemas.

[1] Disponíveis em http://www.cs.virginia.edu/~jmc7tp/resources.php e http://wifo5-03.informatik.uni-mannheim.de/bizer/d2rq/benchmarks/index01.html, acessados em 03/07/2017.

Dataset	Tamanho (MB)	Tabelas	Tuplas	RIR
Mondial	9	28	17.115	104
IMDb	516	5	1.673.074	4
Wikipedia	550	6	206.318	5
DBLP	40	6	878.065	6
TPC-H	876	8	2.389.071	11

Tabela 1 – Características dos Datasets utilizados nos experimentos.

5.1.3 *Query Sets*

Da mesma forma que os *datasets*, foram utilizados em nossos experimentos os mesmos três conjuntos de consultas, ou *Query Sets*, utilizados pelo Lathe. São eles: *Coffman-Weaver, SPARK* e *INEX*. O *Coffman-Weaver* [Coffman e Weaver 2010], aplica um conjunto de consultas para os *datasets* IMDb, Mondial e Wikipedia. O *SPARK* [Luo et al. 2007], foi aplicado no IMDb, Mondial e DBLP, e o IMEX [INEX 2011] somente no IMDb.

No total, foram utilizadas nos experimentos, 218 consultas pré-definidas. Uma visão geral dos *query sets* é apresentada na Tabela 2.

Dataset	Número de Consultas			
	Coffman-Weaver	Spark	IMEX	Total
IMDB	42	22	14	78
Mondial	42	35	-	77
Wikipedia	45	-	-	45
DBLP	-	18	-	18
Total	129	75	14	218

Tabela 2 – Visão geral do conjunto de consultas utilizados nos experimentos.

Um parâmetro importante, que impacta na geração de *Candidate Networks* - CNs, é o número de palavras-chave utilizadas nas consultas, conforme comentado na sessão 3.1.2. Na Tabela 3, apresentamos o número máximo e a média de palavras-chave que constam nos *query sets* utilizados nos experimentos. Considerando todos os *query sets* a média de palavras-chave utilizada nas consultas é 2,17 e o máximo 4, em geral esses são os números de palavras-chave normalmente utilizadas nas consultas feitas por usuários.

Podemos observar que não foram apresentados os dados do *query sets* utilizados no *dataset* TPC-H [2], isso ocorre porque esse *dataset* é utilizado no mercado para testes de performance e escalabilidade e os dados contidos no mesmo são gerados automaticamente. Para este *dataset* foram utilizados *query sets* com consultas geradas a

[2] As ferramentas para geração do *dataset* TPC-H podem ser obtidas a partir do endereço http://www.tpc.org/tpch/. Acesso em 27/02/2018.

Dataset	Número de Palavras-Chave					
	Coffman-Weaver		Spark		INEX	
	Máx	Méd	Máx	Méd	Máx	Méd
IMDB	4,00	2,36	3,00	2,31	4,00	2,42
Mondial	3,00	1,52	3,00	2,17	-	-
Wikipedia	4,00	2,22				
DBLP			4,00	2,77		

Tabela 3 – Números máximos e médios de palavras-chave nas consultas.

partir de palavras-chave aleatórias, em um número muito maior do que o apresentado na Tabela 3, somente para testes de performance e escalabilidade.

5.1.4 Gabaritos

Para verificar a eficiência na geração, classificação e a avaliação de CNs, os experimentos com sistemas R-KwS, baseados em *Schema Graphs*, em geral utilizam um conjunto de CNs e JNTs relevantes esperadas para cada consulta por palavras-chave testada, ou seja um gabarito com os conjuntos de resultados esperados.

No caso do *Coffman* [Coffman e Weaver 2010] além do *query set*, o autor também forneceu um gabarito. Portanto, no Lathe, foram comparadas as JNTs geradas pelas CNs com os dados contidos nesse gabarito para verificar a sua eficiência.

Já no caso dos *query sets SPARK* e *INEX*, no Lathe o gabarito foi gerado manualmente, onde para cada consulta foram geradas todas as CNs, utilizando o algoritmo CNGen [Hristidis e Papakonstantinou 2002], e estas CNs por sua vez foram executadas e identificadas quais eram as JNTs relevantes.

No Lathe-DB, utilizamos nos nossos experimentos os mesmos gabaritos utilizados pelo Lathe.

5.2 Número de CNs relevantes

Assim como no Lathe, nosso trabalho também se baseia na hipótese que o número relevantes de CNs por consulta é muito menor que o número de CNs possíveis de serem geradas. Na Tabela 4, apresentamos o número de CNs relevantes por *query set*, considerando todas as consultas individualmente.

Como podemos observar o número máximo de CNs relevantes é dois, e todas as consultas que tem duas CNs relevantes são encontradas no *dataset* SPARK . No geral a maioria das consultas, 86% no total, tem apenas uma única CN relevante. Outra característica importante é que, além da maioria das consultas gerar apenas uma CN, muitas CNs por sua vez geram apenas uma única JNT como resposta.

Query set	1 CN Relevante		2 CNs relevantes	
Coffman-Weaver	129	100%	-	-
Spark	45	60%	30	40%
INEX	14	100%	-	-
Total	188	86%	30	14%

Tabela 4 – Número de CNs relevantes por *Query set*.

5.3 Experimentos na Geração de CNs

Nesta seção apresentamos os números relacionados aos volumes de dados manipulados pelo Lathe-DB em comparação ao Lathe. Esses números são úteis para avaliar o quão viável é a implementação do método do Lathe no ambiente do SGBD.

5.3.1 Geração de *Query Matches*

Na Tabela 5 , apresentamos os números máximos e médios de *Query Matches* gerados pelo Lathe e pelo Lathe-DB. É possível observar que, em alguns casos, a diferença nos números máximos e médio é grande devido a diferenças na implementação dos algoritmos de geração de *Query Matches* em ambos os trabalhos.

Dataset		Coffman-Weaver		Spark		INEX	
		Max	Med	Max	Med	Max	Med
IMDB	Lathe	69	9,10	45	17,57	123	22,28
	Lathe-DB	30	11,73	21	6,05	31	6,78
Mondial	Lathe	16	4,20	208	23,20	-	-
	Lathe-DB	55	6,85	316	38,68		
Wikipedia	Lathe	36	4,94	-	-	-	-
	Lathe-DB	90	9,87				
DBLP	Lathe	-	-	6	2,00	-	-
	Lathe-DB			18	4,05		

Tabela 5 – Comparação dos números de *Query Matches* gerados.

No Lathe os *matches* são gerados de forma gradual, combinando todos os subconjuntos de *non-free tuple sets* de tamanho 1, em seguida de tamanho 2 e assim por diante. Então o Lathe seleciona todos os subconjuntos de *tuple sets* cujas as palavras-chave formam um conjunto de cobertura mínimo.

Já no Lathe-DB, o algoritmo de geração de *matches*, chamado *genMatches*, conforme detalhado na Sessão 4.4.2, realiza um produto cartesiano com os *non-free tuple sets*, gerando todas as combinações possíveis entre os subconjuntos de *tuple-sets*, o que pode ser uma operação demorada se o número de palavras-chave for grande, porém essa operação ocorre em um tempo que pode ser considerado bom, levando em consideração o número médio de palavras-chave submetidas pelo usuário, conforme demonstrado na Tabela 3.

Em seguida o algoritmo *genMatches* descarta os *matches*, de cada consulta, que não geram um conjunto de cobertura mínimo, ou seja os *matches* cujo o conjunto de termos que o formam não contém todos termos da consulta do usuário. Finalmente é utilizado outro critério de poda, o mesmo utilizado no Lathe, onde são descartados os *matches* cujos o número de *tuple-sets* é maior que o número de palavras-chave [Hearne e Wagner 1973]. Esses processos de poda são fundamentais para diminuir o tempo total de geração das CNs.

Após essas operações de poda, o número de *matches* por consulta é drasticamente reduzido. Mesmo assim o Lathe-DB gera um número maior de *matches* para cada *dataset*, com exceção do *dataset* IMDb, que gerou uma quantidade menor. Essa diferença no *dataset* IMDb ocorreu pela forma como os termos estão distribuídos pelas tabelas, que compõem o *dataset*, e pelo fato de ser o *dataset* com menor número de RIR.

5.3.2 Classificação de *Query Matches*

Nessa seção nos apresentamos os resultado da eficácia do nosso algoritmo de classificação de *Query Matches*, o *qmRank*. Diferente do Lathe, no Lathe-DB a etapa de classificação ocorre antes da etapa de geração das CNs, pois nessa etapa do processo os *Query Matches* já foram gerados e todos os elementos para realizar a classificação já estão disponíveis. O funcionamento do *qmRank* foi detalhado na Seção 4.4.3.

Para avaliar a classificação produzida pelo algoritmo *qmRank*, e comparar com a classificação do algoritmo *CNRank* utilizado pela Lathe, utilizamos a Média de Classificação Recíproca - *Mean Reciprocal Rank* (MRR) [Baeza-Yates e Ribeiro-Neto 1999], a mesma utilizada nos experimentos do Lathe.

Note que, apesar do Lathe-DB ser capaz de classificar as *query matches* antes de gerar as CNs, para utilizar a métrica MRR no nosso experimento é necessário montar as CNs, uma vez que precisamos saber qual é a posição da primeira CN relevante no conjunto de CNs gerada a partir de uma consulta.

Na métrica MRR, dado um conjunto de CNs classificadas, geradas a partir de uma consulta Q, o valor de RR_U é dado por $\frac{1}{Q}$, onde Q é a posição da primeira CN relevante. Então, a MRR é obtida a partir da média de todos RR_U calculados para cada consulta de um *query set*. Intuitivamente, a métrica MRR mede quão perto as CNs relevantes estão da primeira posição do conjunto de CNs classificadas pelos algoritmos *qmRank* e *CNRank*.

Na Figura 31, comparamos os resultados obtidos pelo pelo algoritmo *qmRank* em comparação com o algoritmo *CNRank*, onde cada par de colunas representa a comparação da métrica MMR para cada combinação de *query sets* e *datasets* e, ao final, a média geral da métrica MRR para ambos os algoritmos.

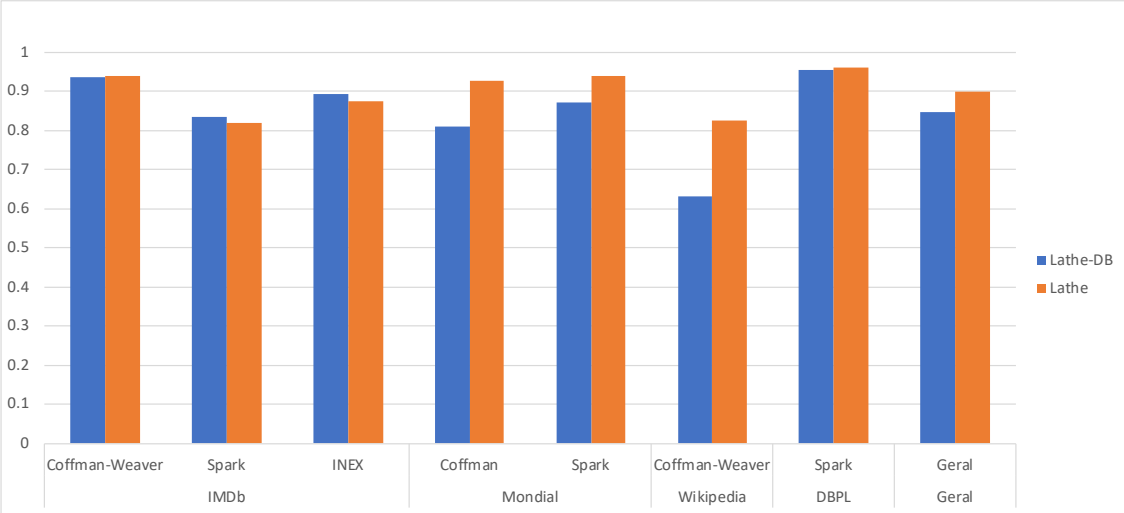


Figura 31 – MRR alcançada pelo Lathe-DB e pelo Lathe para cada *dataset*.

Em seguida, na Figura 32, comparamos o nível de precisão atingido pelo algoritmo *qmRank*, em relação ao *CNRank*, usando a métrica Precisão na Posição K - *Precision at Positon K (P@K)* para cada *dataset* e também, na Figura 33, apresentamos uma média geral da métrica *P@K*.

Na métrica *P@K*, dada um consulta U, o valor de $P_U@K$ é um 1, se a CN relevante aparecer no posição acima de K no conjunto de CNs classificadas, caso contrário o valor de $P_U@K$ é zero. Dessa forma, *P@K* é a média de $P_U@K$, para todo U contido no *query set*.

Nas Figuras 31 e 33 calculamos a média geral dos resultados obtidos, pelas métricas *MRR* e *P@K*, no Lathe-DB e no Lathe, e como podemos observar que os valores estão bem próximos em ambos os trabalhos.

5.3.3 Geração de *Candidate Netwoks*

Para a geração de CNs, o Lathe-DB usa o algoritmo *mountCNs* que se baseia no conceito de *Steiner Trees*, conforme detalhado no Capítulo 4.4.4. Já o Lathe utiliza duas configurações de algoritmos, o *SingleCN* e o *SteinerCN*, e na Figura 34, comparamos os resultados obtidos pelos algoritmos do Lathe com o algoritmo *mountCNs* do Lathe-DB.

Como podemos observar o Lathe-DB gera uma quantidade média menor de CNs por consulta em todas as configurações de *query sets* e *datasets*, com exceção do *dataset* Mondial/Spark.

Notamos também que, apesar do número médio de CNs geradas ser menor, os experimentos que medem a qualidade das CNs geradas, ver Figuras 31 e 33, mostram

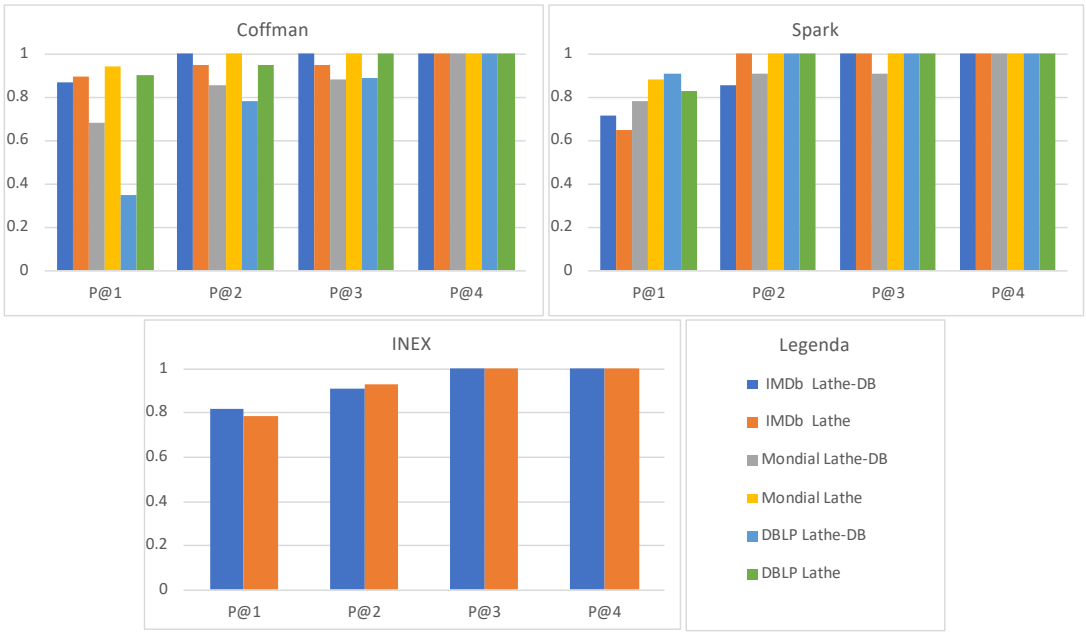


Figura 32 – P@K alcançada pelo Lathe-DB e pelo Lathe, para cada *dataset*.

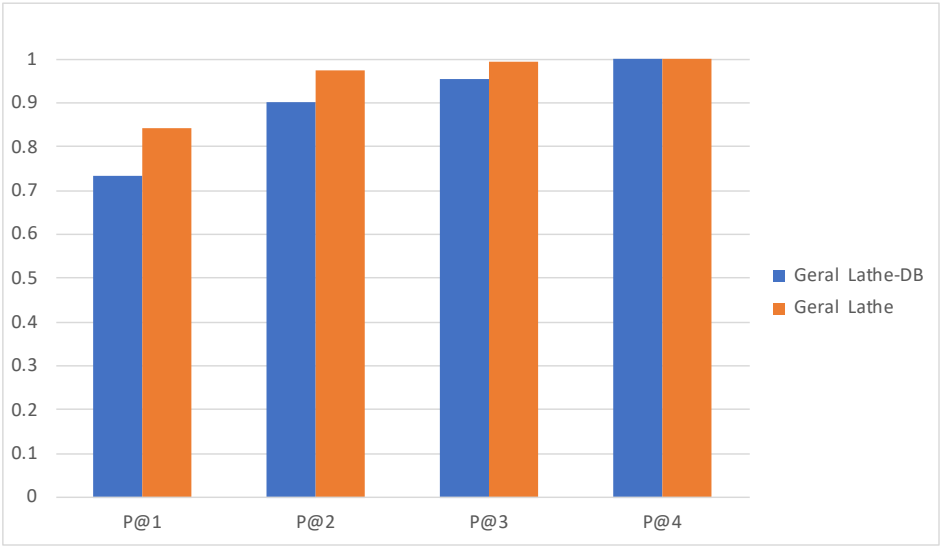


Figura 33 – Comparação de P@K geral alcançada pelo Lathe-DB e Lathe.

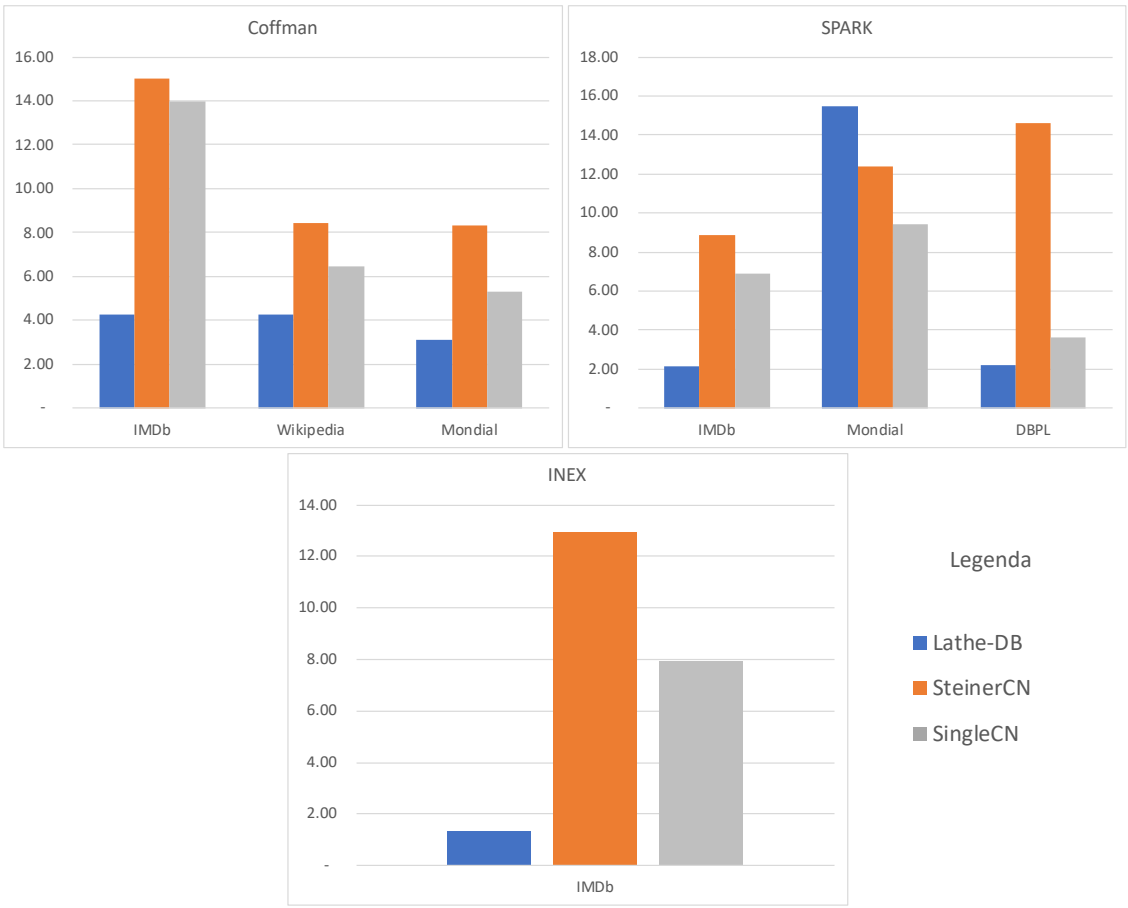


Figura 34 – Número médio de CNs geradas para todos os *query sets* e *datasets*.

que a qualidade das CNs geradas pelo Lathe-DB é tão boa quanto as geradas pelo Lathe.

No próximo experimento comparamos o desempenho do nosso algoritmo de geração de CNs com os resultados obtidos pelo Lathe. O Lathe possui duas versões de seu algoritmo principal: o *MatCNGen-Disk*, chamado de MCG-D, onde a geração de tuplas é feita utilizando o disco; e o *MatCNGen-Men*, chamando MCG-M, que, como o nome sugere, faz todo o processamento em memória. Lembrando que, como o Lathe-DB é implementado no ambiente do SGBD, todo seu processamento se faz em disco.

Na Figura 35 comparamos, para todos os *datasets* e *query sets* os tempos médios de geração de CNs. Cada barra no gráfico representa o tempo médio que o os algoritmos levaram para gerar as CNs, e em cada barra foram separados os tempos gastos para a geração de geração dos *tuples-set* e o tempo para a construção das CNs. Note que no tempo de geração dos *tuple-sets* esta incluso o tempo de geração dos *query matches*.

Como podemos observar o Lathe-DB, por utilizar todos os recursos de desempenho disponíveis no SGBD e pelo fato de precisar processar um número menor de CNs, consegue tempos menores que o Lathe para todas as combinações de *query sets* e *datasets*.

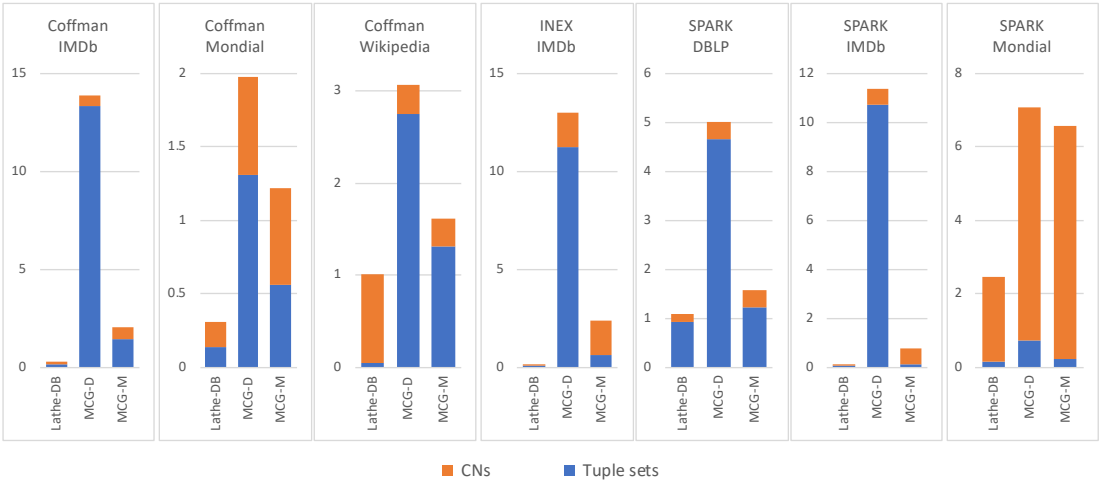


Figura 35 – Tempo médio de geração de CNs para todos os *query sets* e *datasets*.

No próximo experimento analisamos uma conhecida desvantagem dos atuais sistemas R-KwS, o que se refere a escalabilidade dos sistemas com o número alto de palavras-chave nas consultas. De fato esse problema tem sido estudado na literatura [Baid et al. 2010, Markowetz, Yang e Papadias 2007] uma vez que um número alto de palavras-chave na consulta causa um excessivo consumo de memória.

Assim como nos experimentos do Lathe, para cada *dataset* geramos aleatoriamente uma carga de 10 consultas com K palavras-chave, sendo que K varia de 1 a 10. Para cada consulta medimos o tempo gasto para gerar as CNs e, mais uma vez, comparamos com os resultados obtidos pelo Lathe-DB com os resultados do Lathe. Os resultados são apresentados na Figura 36, cada curva corresponde a média do tempo gasto no processamento de 100 consultas para cada valor de K.

Nesse experimento o Lathe leva vantagem, uma vez que o processamento do índice de termos, e demais estruturas de dados, é realizado em memória. Já o Lathe-DB necessita realizar essas leituras em disco a cada nova consulta.

O tempo gasto pelo Lathe-DB é maior em todos os cenários quando o número de palavras-chave aumenta. Porém ainda assim se mostrou capaz de lidar com um grande número de palavras-chave na consulta. Assim como comentado no Lathe, trabalhos anteriores na literatura mostram que o processamento de um grande número de palavras-chave sobre bancos de dados relacionas, possuem tempos de processamento

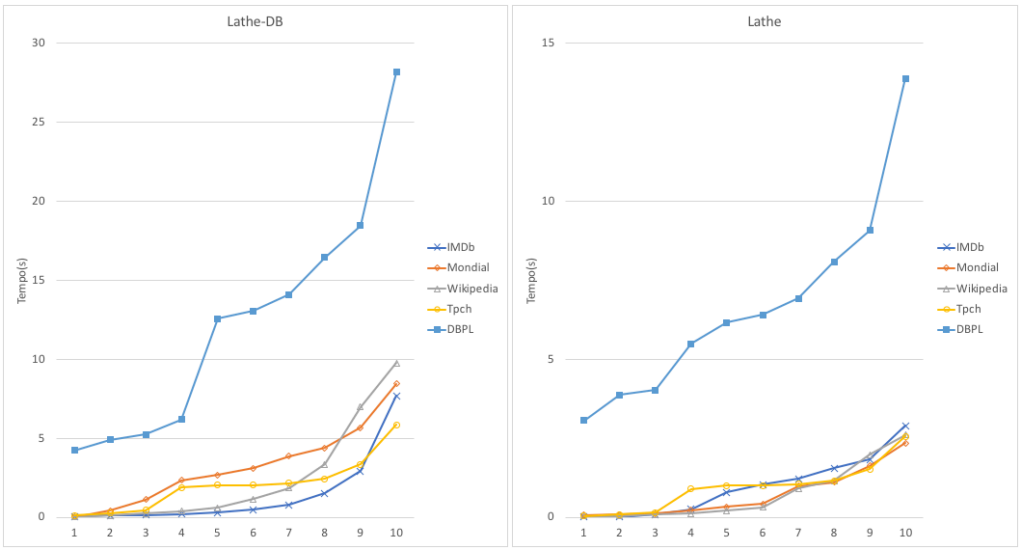


Figura 36 – Tempo médio gasto para gerar as *Candidate Networks* quando variamos o número de palavras-chave para cada *dataset*.

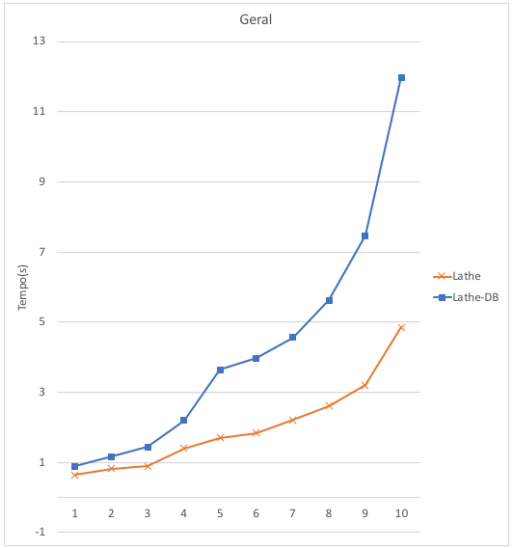


Figura 37 – Tempo médio geral gasto para gerar as *Candidate Networks* quando variamos o número de palavras-chave.

imprevisíveis, e certas consultas levam um tempo tão longo de processamento que falham ao consumir os recursos de memória disponíveis [Baid et al. 2010, Coffman e Weaver 2010]. De fato, a geração de *Candidate Networks* pode ser uma operação custosa independentemente do método utilizado.

6 Conclusão

Este capítulo apresenta as conclusões finais deste trabalho, bem como os que temos planejado para o futuro do Lathe-DB.

6.1 Resultados Obtidos

Neste trabalho, apresentamos o Lathe-DB, um mecanismo de busca por palavras-chave integrado ao SGBD, implementando com sucesso o método do Lathe [Oliveira e Silva 2015, Oliveira e Silva 2017].

Os resultados apresentados no Capítulo 5 mostram a viabilidade técnica de se adotar esse mecanismo integrado ao SGDB ao comparar os resultados obtidos com os principais métodos propostos na literatura.

Demonstramos sua viabilidade e superamos os desafios de utilizar somente os recursos oferecidos como padrão pelo SGBD PostgreSQL, dessa forma esperamos ter contribuído da adoção desta funcionalidade pela comunidade de desenvolvedores de sistemas.

6.2 Trabalhos Futuros

Apesar dos bons resultados apresentados com o Lathe-DB, é possível realizar novas pesquisas para melhorar o desempenho e precisão dos resultados fornecidos pelo nosso Mecanismo de Busca por Palavras-Chave em SGBDs Relacionais.

Primeiramente planejamos criar um mecanismo para atualizar dinamicamente o índice de termos, com o cuidado de fazer o máximo para não alterar a estrutura banco de dados alvo e impactar em seu funcionamento. Essa melhoria é importante para o uso do Lathe-DB em sistemas com um grande volume de atualizações.

Buscar técnicas para melhoria da performance geral do Lathe-DB, em especial técnicas para otimizar o tempo de processamento para consultas com um grande número de palavras-chave.

Outra técnica que estamos considerando aplicar é a de Normalização Morfológica, ou *Stemming*, durante o processo de indexação. Em alguns casos, torna-se interessante eliminar as variações morfológicas dos termos através da remoção dos prefixos e o sufixos e adicionar radicais resultantes ao índice. Além disso características de gênero, número e grau também são eliminadas dos termos, o que pode reduzir substancialmente o tamanho do índice [Riloff 1995].

Além da técnica de Normalização Morfológica consideramos ainda o uso de Funções Fonéticas, para que o Lathe-DB seja capaz de realizar a busca de termos que possuem o mesmo som, como por exemplo "Luiz"e "Luís", ou ainda termos que estejam grafados incorretamente, como os termos "Exceder"e "Esceder". Funções fonéticas estão implementadas nativamente na maioria dos SGBDs comerciais e adicionar essa funcionalidade a um sistema R-KwS ainda não foi considerada na maioria dos principais trabalhos.

Também consideramos tratar a identificação de Termos Compostos, uma vez que muitos termos tem significado diferente quando utilizados em conjunto, como por exemplo "Palavra Chave"ou "Processo Cível". Essa identificação pode ser feita com a utilização de um dicionário ou com a identificação de termos que co-ocorrem com muita frequência [Wives 2002].

Finalmente, planejamos portar o Lathe-DB para os principais SGBDs do mercado, a fim de aumentar a sua adoção pela comunidade.

Referências

ADITYA, B. et al. BANKS: Browsing and keyword searching in relational databases. In: *Proc. of the 28th Intl. Conf. on VLDB*. [S.l.: s.n.], 2002. p. 1083–1086. Cited on page 34.

AGRAWAL, R.; IMIELINSKI, T.; SWAMI, A. Mining association rules between sets of items in large databases. In: *IN: PROCEEDINGS OF THE 1993 ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, WASHINGTON DC (USA*. [S.l.: s.n.], 1993. p. 207–216. Cited on page 59.

AGRAWAL, S.; CHAUDHURI, S.; DAS, G. DBXplorer: A system for keyword-based search over relational databases. In: *Proc. of the 18th Intl. Conf. on Data Engineering*. [S.l.: s.n.], 2002. p. 5–16. Cited 3 times on pages 23, 32 e 34.

AGRAWAL, S. et al. Automated ranking of database query results. In: *Conf. on Innovative Data Systems Research*. [S.l.: s.n.], 2003. Cited on page 23.

BAEZA-YATES, R. A.; RIBEIRO-NETO, B. A. *Modern Information Retrieval*. [S.l.]: Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. Cited on page 77.

BAEZA-YATES, R. A.; RIBEIRO-NETO, B. A. *Modern Information Retrieval - the concepts and technology behind search, Second edition*. [S.l.]: Pearson Education Ltd., Harlow, England, 2011. Cited on page 45.

BAID, A. et al. Toward scalable keyword search over relational data. *Proc. VLDB Endow.*, v. 3, n. 1-2, p. 140–149, 2010. Cited 6 times on pages 32, 33, 37, 44, 81 e 83.

COFFMAN, J.; WEAVER, A. C. A framework for evaluating database keyword search strategies. In: *CIKM*. [S.l.: s.n.], 2010. p. 729–738. Cited 4 times on pages 33, 44, 73 e 83.

COFFMAN, J.; WEAVER, A. C. Relational keyword search benchmark. In: . [S.l.: s.n.], 2010. Http://www.cs.virginia.edu/ jmc7tp/resources.php. Cited 6 times on pages 31, 33, 38, 73, 74 e 75.

COFFMAN, J.; WEAVER, A. C. Structured data retrieval using cover density ranking. In: *Proc. of the 2nd Intl. Workshop on KwS on Structured Data*. [S.l.: s.n.], 2010. p. 1:1–1:6. Cited on page 32.

DING, B. et al. Finding top-k min-cost connected trees in databases. In: *Proc. of the 23rd Intl. Conf. on Data Engineering*. [S.l.: s.n.], 2007. p. 836–845. Cited on page 32.

DREYFUS, S. E.; WAGNER, R. A. The steiner problem in graphs. *Networks*, v. 1, n. 3, p. 195–207, 1971. Disponível em: <http://dx.doi.org/10.1002/net.3230010302>. Cited on page 24.

FAN, J.; LI, G.; ZHOU, L. Interactive sql query suggestion: Making databases user-friendly. In: IEEE. [S.l.], 2011. Cited 2 times on pages 23 e 36.

GOLDMAN, R. et al. Proximity search in databases. In: *VLDB*. [S.l.: s.n.], 1998. Cited on page 67.

HAN, J.; PEI, J.; YIN, Y. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 29, n. 2, p. 1–12, maio 2000. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/335191.335372>. Cited on page 59.

HEARNE, T.; WAGNER, C. Minimal covers of finite sets. *Discrete Math.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 5, n. 3, p. 247–251, jul. 1973. ISSN 0012-365X. Disponível em: <http://dx.doi.org/10.1016/0012-365X(73)90141-6>. Cited on page 77.

HRISTIDIS, V.; GRAVANO, L.; PAPAKONSTANTINOU, Y. Efficient IR-style keyword search over relational databases. In: *Proc. of the 29th Intl. Conf. on VLDB*. [S.l.: s.n.], 2003. p. 850–861. Cited 3 times on pages 23, 32 e 35.

HRISTIDIS, V.; PAPAKONSTANTINOU, Y. DISCOVER: keyword search in relational databases. In: *Proc. of the 28th Intl. Conf. on VLDB*. [S.l.: s.n.], 2002. p. 670–681. Cited 12 times on pages 23, 24, 29, 30, 32, 33, 34, 35, 37, 42, 43 e 75.

INEX. *INitiative for the Evaluation of XML Retrieval*. 2011. Https://inex.mmci.uni-saarland.de/about.html. Cited 2 times on pages 73 e 74.

LI, G. et al. Providing built-in keyword search capabilities in rdbms. In: SPRINGER-VERLAG. *Proceedings of the twelfth international conference on Information and knowledge management*. [S.l.], 2010. Cited 4 times on pages 23, 24, 25 e 36.

LUO, Y. et al. SPARK: top-k keyword query in relational databases.technical report unsw-cse-tr-0708. In: *Proc. of the 2007 ACM SIGMOD Intl. Conf. on Manag. of data*. [S.l.: s.n.], 2007. Cited 5 times on pages 32, 44, 46, 73 e 74.

MACULA, A. J. Covers of a finite set. *Mathematics Magazine*, n. 67(2), p. 141–144, 1994. Cited on page 42.

MARKOWETZ, A.; YANG, Y.; PAPADIAS, D. Keyword search on relational data streams. In: *Proc. of the 2007 ACM SIGMOD Intl. Conf. on Management of Data*. [S.l.: s.n.], 2007. p. 605–616. Cited 3 times on pages 32, 33 e 81.

MESQUITA, F. et al. LABRADOR: Efficiently publishing relational databases on the web by using keyword-based query interfaces. *Inf. Process. Manage.*, v. 43, n. 4, p. 983–1004, 2007. Cited 4 times on pages 23, 35, 45 e 66.

NANDI, A.; JAGADISH, H. V. Qunits: queried units in database search. In: *CIDR 2009, Fourth Biennial Conf. on Innovative Data Systems Research*. [S.l.: s.n.], 2009. Cited on page 44.

OLIVEIRA, P.; SILVA, A. S. da. *Lathe: Light-Weight Keyword query Processing over Multiple Databases*. 2015. 0 p. Manuscrito não publicado. Cited 11 times on pages 23, 24, 26, 28, 34, 36, 37, 47, 59, 73 e 85.

OLIVEIRA, P.; SILVA, A. S. da; MOURA, E. S. de. Ranking candidate networks of relations to improve keyword search over relational databases. In: *ICDE*. [S.l.: s.n.], 2015. p. To appear. Cited 5 times on pages 26, 37, 45, 60 e 73.

OLIVEIRA, P. S. de; SILVA, A. S. da. *Generation and Ranking of Candidate Networks of Relations for Keyword Search over Relational Databases*. Tese (Doutorado) — Universidade Federal do Amazonas, Manaus - Amazonas, 3 2017. Cited 11 times on pages 23, 24, 26, 28, 34, 36, 37, 47, 59, 73 e 85.

RILOFF, E. Little words can make big difference for text classification. In: *ANNUAL INTERNATIONAL ACM-SIGIR CONFERENCE ONRESEARCH AND DEVELOPMENT IN INFORMATION RETRIEVAL(SIGIR'95)*. [S.l.: s.n.], 1995. Cited on page 85.

TAKAHASHI; MATSUYAMA. An approximate solution for the steiner problem in graphs. *Math. Japonica*, v. 24, p. 573–577, 1980. Cited 3 times on pages 36, 43 e 62.

WIVES, L. K. *Tecnologias de Descoberta de Conhecimentos em Textos Aplicadas à Inteligência Competitiva*. 2002. Cited on page 86.

ZAKI, M. J. Scalable algorithms for association mining. *TKDE*, v. 12, n. 3, p. 372–390, 2000. Cited 2 times on pages 39 e 59.