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

Generation and Ranking of Candidate Networks of Relations for Keyword Search over Relational Databases

by

Péricles Silva de Oliveira

Manaus – Amazonas

March 2017

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

Péricles Silva de Oliveira

Generation and Ranking of Candidate Networks of Relations for Keyword Search over Relational Databases

Tese apresentada ao Programa de Pós Graduação em Informática da Universidade Federal do Amazonas, como requisito parcial para a obtenção do título de Doutor em Informática, área de concentração em Banco de Dados e Recuperação da Informação.

orientador: Prof. Dr.Altigran Soares da Silva

Manaus – Amazonas

March 2017

Ficha Catalográfica

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

O48g

Oliveira, Péricles Silva de
Generation and Ranking of Candidate Networks of Relations for Keyword / Péricles Silva de Oliveira. 2017
78 f.: il. color; 31 cm.

Orientador: Altigran Soares da Silva
Tese (Ciência da Computação) - Universidade Federal do Amazonas.

1. keyword-search. 2. match graph. 3. relational database. 4. ranking. 5. candidate networks. I. Silva, Altigran Soares da II. Universidade Federal do Amazonas III. Título

Folha de Aprovação

Dedicated to my family: Edilza Oliveira (my wife), Luise Oliveira (my daughter) and Cristhian Oliveira (my son).

In Memory of my mother Maria de Jesus, and my second mother Edith Cordeiro.

Abstract

Several systems proposed for processing keyword queries over relational databases rely on the generation and evaluation of Candidate Networks (CNs), i.e., networks of joined database relations that, when processed as SQL queries, provide a relevant answer to the input keyword query. Although the evaluation of CNs has been extensively addressed in the literature, problems related to efficiently generating meaningful CNs have received much less attention. To generate useful CNs is necessary to automatically locating, given a handful of keywords, relations in the database that may contain relevant pieces of information, and determining suitable ways of joining these relations to satisfy the implicit information need expressed by a user when formulating her query. In this thesis, we present two main contributions related to the processing of Candidate Networks. As our first contribution, we present a novel approach for generating CNs, in which possible matchings of the query in database are efficiently enumerated at first. These *query matches* are then used to guide the CN generation process, avoiding the exhaustive search procedure used by current state-of-art approaches. We show that our approach allows the generation of a compact set of CNs that leads to superior quality answers, and that demands less resources in terms of processing time and memory. As our second contribution, we initially argue that the number of possible Candidate Networks that can be generated by any algorithm is usually very high, but that, in fact, only very few of them produce answers relevant to the user and are indeed worth processing. Thus, there is no point in wasting resources processing useless CNs. Then, based on such an argument, we present an algorithm for ranking CNs, based on their probability of producing relevant answers to the user. This relevance is estimated based on the current state of the underlying database using a probabilistic Bayesian model we have developed. By doing so we are able do discard a large number of CNs, ultimately leading to better results in terms of quality and performance. Our claims and proposals are supported by a comprehensive set of experiments we carried out using several query sets and datasets used in previous related work and whose results we report and analyse here.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last decade, many researchers have proposed methods to enable keyword searches over relational databases. Their goal is to allow naive users to retrieve information without any knowledge about schema details or query languages. Empowering users to search relational databases using keyword queries is a challenging task. In particular, the information sought often spans multiple tuples and tables, according to the schema design of the underlying database. Thus, systems that process keyword-based queries over relational databases, commonly called *Relational Keywords Search Systems* or *R-KwS* systems, face the challenging task of automatically determining, from a handful of keywords, that pieces of information to retrieve and how these pieces can be combined to provide a relevant answer to the user.

Current R-KwS systems fall in one of two distinct categories: systems based on *Schema Graphs* and systems based on *Data Graphs*. Systems in the first category are based on the concept of Candidate Networks (CNs), that are networks of joined database relations that are used to generate SQL queries whose results provide an answer to the input keyword query. This approach was proposed in DBXplorer [Agrawal et al., 2002] and DISCOVER [Hristidis and Papakonstantinou, 2002], and was later adopted by a number of other systems, such as Efficient [Hristidis et al., 2003], SPARK [Luo et al., 2007a], CD [Coffman and Weaver, 2010c], Min-cost [Ding et al., 2007], S-KwS [Markowetz et al., 2007], KwS-F [Baid et al., 2010], PandaSearch [Huang et al., 2015]. Systems in this category take advantage of the basic functionality of the underlying RDBMS by producing appropriate SQL joint queries to retrieve answers relevant to keyword queries posed by users.

Systems in the second category are based on structures called *Data Graph*, whose nodes represent tuples associated with the keywords they contain and edges connect these tuples based on referential integrity constraints. In this approach, adopted by a number of systems, including BANKS [Aditya et al., 2002], Bi-directional [Kacholia et al., 2005], BLINKS [He et al., 2007],

ClearMap [Bao et al., 2015] and Effective [Liu et al., 2006], results of keyword queries are computed by finding subtrees in a data graph that minimize the distance between nodes matching the given keywords. Data graphs use schema information and, thus, are not tied to the relational model.

Besides the relational model, there are several works in the literature that address the processing of keyword queries over XML data. Initially, most of the research efforts was focused on keyword queries targeted to stored collections of XML documents [Le and Ling, 2016; Liu and Chen, 2011; Liu and Cher, 2008; Sun et al., 2007; Tian et al., 2011; Vagena et al., 2007; Xu and Papakonstantinou, 2008; Zhou et al., 2010]. Later on, a few methods were proposed to handle streams of XML documents [Barros et al., 2016, 2010; da C. Hummel et al., 2011; Vagena and Moro, 2008].

In this thesis we present contributions and results related to the processing of keyword queries over relational databases. More specifically, we aim at improving systems based on Schema Graphs. In this context, we propose novel approaches for the problems of generating and ranking Candidate Networks, as described below.

1.1 Match-Based Candidate Network Generation

As our first main contribution, we present a novel approach for generating Candidate Networks. In a nutshell, our approach aims at pruning the exponential number of combinations of relations subsets that arise during the CN generation process. Our motivation is making CN-based R-KwS systems efficient and scalable for using in on-line settings.

Indeed, it is known that, for certain queries, current systems can take too long to produce answers, and for others they may even fail to return results (e.g., by exhausting memory) [Baid et al., 2010; Markowetz et al., 2007].

Interestingly, since its definition in Hristidis and Papakonstantinou [2002], the CN generation problem has been ill studied in the literature. In fact, most of the existing work [Agrawal et al., 2002; Coffman and Weaver, 2010c; Ding et al., 2007; Hristidis et al., 2003; Luo et al., 2007a], has focused on the problem of CN evaluation instead, adopting the CN generation algorithm proposed in Hristidis and Papakonstantinou [2002], called *CNGen*, as default. One of the few exceptions is R-KwS-F [Baid et al., 2010], that proposes important practical pruning strategies to deal with a potentially explosive number of CNs, but that does not address the generation process itself.

We claim that a major issue with the current approach to generate CNs is that it requires exhaustively exploring the full, and sometimes explosive, combination of the multiple keywords

occurrences in the database and the multiple ways these occurrences can be joined. The original algorithm for generating CNs, *CNGen*, that is used as default by most CN-based R-KwS systems, works by first locating subsets of relations in that keywords of the query occur. Then, the algorithm executes an exhaustive search procedure over the graph of the database schema to generate combinations of these subsets in the form of join trees that may fulfill the input query.

The approach we propose here, called *Match-Based Candidate Network Generation*, or *MatCNGen* consists of first enumerating the possible ways that query keywords can be matched in the database to generate query answers. Then, each of these *query matches* is used to induce subgraphs in the database schema graph. Finally, a CN generation algorithm runs over each induced subgraph individually. We argue that this strategy drastically reduces time required to generate CNs and we present several experimental results to support this claim.

For properly implementing *MatCNGen*, we proposed algorithms to efficiently execute each of its main steps. The **QMGen** algorithm was developed for efficiently combining tuple-sets, forming *query matches*. This algorithm leverages string combination properties to prune the space of possible combinations. For carrying out the searching for joins among tuple-sets in the database schema graph to ultimately obtain Candidate Networks, we propose two alternative algorithms. The first, we call **SingleCN**, is based on the well-known breadth-first traversal algorithm. The second, we call **SteinerCN**, is based on the concept of Steiner Trees [Dreyfus and Wagner, 1971], that generalize graph connectivity concepts such as shortest-paths and minimal spanning trees. We developed the **TSFind** algorithm to find subsets of relations that contain the keywords of the query, called here *tuple-sets*. This algorithm was develop to reduce database operations required for this task to a few sequential disk accesses.

All of these algorithms comprise original contributions of this work. They will be detailed in the text and results of experiments we carried out with them will be reported in the experiments chapters.

1.2 Ranking Candidate Networks

During our work with the development of *MatCNGen*, we notice that, depending on the query and the target database, there can be a large number of Candidate Networks generated. For instance, the experimental query workload we use in our experiments includes queries from the original CN generation algorithm, CNGen [Hristidis and Papakonstantinou, 2002], obtains hundreds of CNs. Processing a large number of CNs, is of course, time-demanding and resource-consuming. Moreover, the quality of the answers produced by CN evaluation may be compromised when a large number of CNs is processed.

We claim that, although the number of possible Candidate Networks can be very high, only very few of them produce answers relevant to the user and are indeed worth processing. This claim is in line with observations made by other researchers who found that the number of relevant answers to keyword queries is often very small and that, in many cases, there is only one relevant answer to return [Baid et al., 2010; Coffman and Weaver, 2010a; Luo et al., 2007a; Nandi and Jagadish, 2009]. It follows that, if only a few answers are relevant, then only a few CNs need to be evaluated to produce them. We also observed this trend in queries of different workloads we used in experiments we carried out and report here. This is significant, since these workloads have been proposed and used in previous studies on R-KwS in the literature. In fact, we verified that, in all queries in these workloads, no more than two Candidate Networks are needed to produce relevant answers. This is a drastic reduction, if we consider that the number of CNs generated often ranges from tens to many hundreds.

Making this claim explicit and showing experimental data to support it is also a contribution we made in this thesis. An implication of this claim is the need for methods to assess the relevance of Candidate Networks, so that only those deemed relevant might be evaluated.

With this goal in mind, we present in this thesis an approach for ranking Candidate Networks, called *CNRank*,based on their probability of producing relevant answers to the user. Specifically, we present a probabilistic ranking model that uses a Bayesian belief network to estimate the relevance of a Candidate Network given the current state of the underlying database. A score is assigned to each generated Candidate Network so that only a few CNs with the highest scores are evaluated. In addition, we also show how this ranking process can be carried out efficiently using a simple inverted index. This approach, the model and the ranking algorithm also comprise contributions we offer in our work.

Using the proposed approach, we performed a comprehensive set of experiments using query workloads also used in R-KwS experiments previously presented in the literature. By comparing our results with those obtained with other representative methods on the same tasks, we could observe that our approach had a considerable positive impact, not only on the performance of processing keyword queries, but also on the quality of the answers produced by CN evaluation and JNT ranking algorithms. For instance, when we coupled our CN ranking algorithm with two well-known CN evaluation algorithms, namely, Hybrid [Hristidis et al., 2003] and Skyline Sweeping [Luo et al., 2007a], the results they deliver were twice as precise, according to widely-accepted metrics, in compared with the results they provide without our algorithm. As these evaluation algorithms received less CNs to process, they also run much faster. In addition, we have experimentally shown that our ranking model is very precise: for all the queries we tested, it was able to place the relevant CNs among the top-4 in the ranking produced. Showing experimental evidences of the impact of our approach in the performance and the quality of the answers produced by R-KwS systems is also a contribution we make in this work.

Our method for ranking Candidate Networks was first published in a full paper accepted for the IEEE 2015 International Conference on Data Engineering (ICDE) [de Oliveira et al., 2015].

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 reviews the related literature, notation and terminology used in the field of Relational Keywords Search Systems based on Schema Graphs. Chapter 3 overviews our two main contributions and the components of a new architecture we propose for R-KwS systems. Chapters 4 formalizes our strategy for the problem of generating CNs and presents our algorithm for efficiently combining tuple-sets. Our two alternative CN generation algorithms are described in Chapter 5. Chapters 6 presents our algorithm to find subsets of relations that contain the keywords of the query (tuple-sets). In Chapter 7, we present the details of our model and algorithm for ranking Candidate Networks. Chapters 8, 9 and 10 report the results of experiments we have conducted with implementations of our algorithms, comparing these results with those obtained with representative baselines. Finally, Chapter 11 presents conclusions we have reached and outlines some directions for future work.

Chapter 2

Background and Related Work

In our research we focus on systems based on Schema Graphs, since we assume that the data we want to query are stored in a relational database and we want to use a RDBMS capable of processing SQL queries. This section reviews the general approach adopted by these systems and discuss previous related work in the literature. We begin by providing an overview of Schema Graph R-KwS Systems. Next, we review a number of important concepts and the terminology introduced in Hristidis and Papakonstantinou [2002], which we follow in this thesis. Only for convenience, some definitions are re-stated with slight modifications with respect to their original versions. For a more broad coverage of the literature in the topic, the interested reader may want to read a comprehensive survey by Yu et al. [2010].

2.0.1 Schema Graph R-KwS Systems

Figure 2.1 presents the main architecture and functioning of a typical R-KwS System based on schema graphs, which we describe below. Initially, an keyword query is submitted by an user, for instance, using a single text box (1). A keyword query searches for interconnected tuples that contain the given keywords. A tuple contains a keyword if a text attribute of the tuple contains the keyword. With the keywords in hand, the system looks for subsets of relations that contain the keywords from the queries. These subsets, called *Tuple-Sets*, are then retrieved from the database (2); Next, these tuple-sets are used to generate *Candidate Networks* [Agrawal et al., 2002; Hristidis and Papakonstantinou, 2002] (CNs) (3). CNs are relational algebra expressions that join relations whose tuples contain the keywords being sought. In other words, each CN describes how to produce potential answers to the keyword query entered. Besides the tuple-sets generated in the previous step, generating CNs also requires information on referential integrity constraints (RIC) taken from the database schema. In general, since there are many different ways of joining relations that store the tuples containing the keywords, many different CNs can potentially be generated. In practice, however, only a few of them are useful for producing

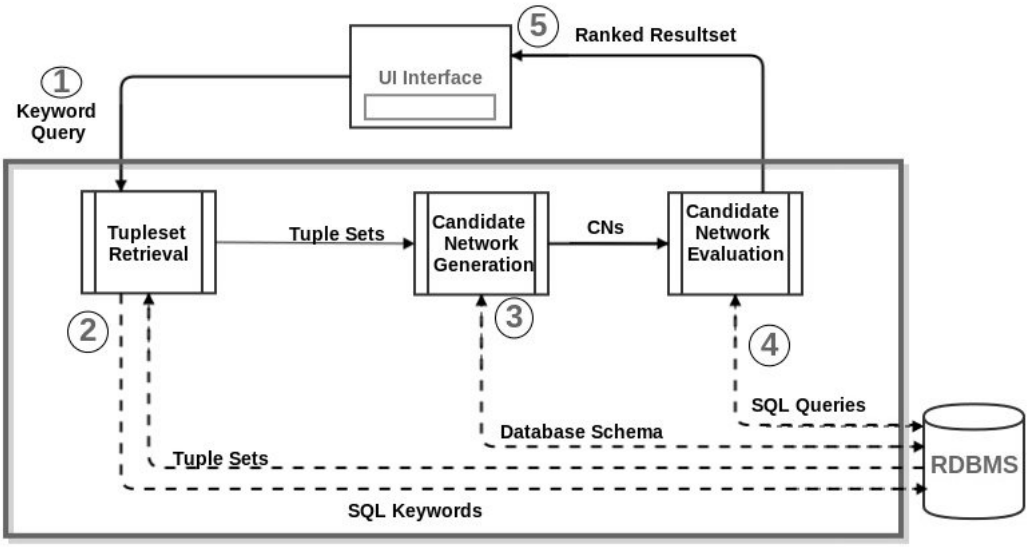


FIGURE 2.1: Overview of a typical RKwS system

plausible answers. The most well-know algorithm for generating CNs, called *CNGen*, was proposed in Hristidis and Papakonstantinou [2002]. In the next step (4), the CNs generated are evaluated in order to get answers from the database to fulfill the input query. In this context, answers are *Joining Networks of Tuples* (JNTs) [Hristidis and Papakonstantinou, 2002], that is, trees composed by joined tuples that either contain the keywords or associate tuples that contain the keywords. Many different algorithms have been proposed to evaluate CNs [Agrawal et al., 2002; Hristidis and Papakonstantinou, 2002; Markowetz et al., 2007]. In particular, in state-of-the-art systems [Coffman and Weaver, 2010c; Hristidis et al., 2003; Luo et al., 2007a], only top-K JNTs are retrieved, which requires them to be ranked using IR style score functions. The top-K answers are then presented to the user (5).

The main motivation for ranking JNTs in systems like Efficient [Hristidis et al., 2003], SPARK [Luo et al., 2007a] and CD [Coffman and Weaver, 2010c], is to avoid the multi-query optimization problem that arises when all CNs are to be evaluated, as occurs in DISCOVER [Hristidis and Papakonstantinou, 2002]. The efficiency and scalability problems in CN evaluation are addressed in a different way in KwS-F [Baid et al., 2010]. Their approach consists of two steps. First, a limit is imposed on the time the system spends evaluating CNs. After this limit is reached, the system must return a (possibly partial) top-K JNTs result. Second, if there are CNs yet to be evaluated, these CNs are presented to the user by means of query forms, so the user can select one of the forms and the system evaluates the corresponding CN. In Baid et al. [2010], the authors report a number of experimental results on the effectiveness and feasibility of the proposed approach. Unfortunately, no results on the quality of the results obtained are reported. Although our experiments have shown that our proposed method generates, without quality loss, fewer CNs than CNGen, it could be naturally combined with KwS-F if necessary.

In Markowetz et al. [2007], the authors present a strategy for ordering the internal nodes of the CNs generated by CNGen aiming to detect possible duplicated CNs. This can reduce the number of CNs handled by the systems, but still requires that all CNs, duplicate or not, be generated. As we will discuss, the approach we propose here avoids duplicated CNs by construction, using the concept of *query match* which we will properly introduce later.

In Coffman and Weaver [2010a] proposed a framework for evaluating R-KwS systems and reported the results of applying this framework over three representative standardized datasets they built, namely *Mondial*, *IMDb* and *Wikipedia*, along with respective query workloads. The authors compare nine state-of-the-art R-KwS systems, evaluating them in many aspects related to their effectiveness and performance. An important conclusion they report is that, in terms of effectiveness, no single system tested performed best across all datasets/query sets. So far, this is the only study in the literature to address the qualitative evaluation of R-KwS systems. In our experiments we use datasets, query sets and results from this paper.

2.1 Basic Concepts and Terminology

In the following, we review several basic concepts related to R-KwS systems. We rely on the same terminology and concepts introduced in Hristidis and Papakonstantinou [2002]. In particular, we use the definitions for Joining Network of Tuples, Minimal Total Joining Network of Tuples, Tuple-Sets, Joining Network of Tuple-Sets and Candidate Networks. Only for convenience, some definitions are re-stated with slight modifications with respect to Hristidis and Papakonstantinou [2002] work.

As in Hristidis and Papakonstantinou [2002], consider a schema graph G representing a relational schema, where vertices correspond to relations and edges correspond to referential integrity constraints between relations. For the discussion that follows, the directions of referential constraints are not important, so we consider an undirected version G_u of G[1].

Definition 2.1. A **Joining Network of Tuples (JNT)** j is a tree of tuples where for each pair of adjacent tuples $t_i, t_j \in j$, where t_i and t_j are tuples of relations R_i and R_j, respectively, there is an edge $\langle R_i, R_j \rangle$ in G_u and $(t_i \bowtie t_j) \in (R_i \bowtie R_j)$.

Definition 2.2. Given a set $Q = \{k_1, \ldots, k_n\}$ of keywords, a JNT j is a **Minimal Total Joining Network of Tuples (MTJNT)** for Q if it is both **total**, that is every keyword k_i is contained in at least one tuple of j, and **minimal**, that its, any JNT j' that results from removing any tuple from j is not total.

[1] Without loss of generality, we also assume the same simplifications as Hristidis and Papakonstantinou [2002], that is, the attributes involved in referential integrity constraints have the same name, there are no self loops or parallel edges in the schema graph, and no set of attributes of any relation is both a primary key and a foreign key for two other relations

Definition 2.3. A **Keyword Query** is a set Q of keywords whose result is the set M of all possible MTJNT for the keywords in Q over some set of relations $\{R_1, \ldots, R_m\}$.

Definition 2.4. Let Q be a keyword query and let K a subsets of Q. Also, let R_i be a relation. A **Tuple-Set** from R_i over K is given by

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

where $\mathcal{W}(t)$ gives the set of terms (words) in t. If $K = \emptyset$, the tuple-set is said to be a *free tuple-set* and it is denoted by $R_i^{\{\}}$.

According to Definition 2.4, the tuple-set R_i^K contains the tuples of R_i that contain all terms of K and no other keywords from Q.

Definition 2.5. A **Joining Network of Tuple-Sets** J is a tree of tuple-sets where for each pair of adjacent tuple-sets R_i^K , R_j^M in J there is an edge $\langle R_i, R_j \rangle$ in G_u.

Definition 2.6. Given a keyword query $Q = \{k_1, \ldots, k_n\}$, a **Candidate Network** C is a joining network of tuple-sets, such that there is an instance I of the database that has a MTJNT $M \in C$ and no tuple $t \in M$ that maps to a free tuple-set $F \in C$ contains any of the keywords from Q.

Notice that, by Definition 2.6, the answer produced by a CN must be a MTJNT, that is, a set of total and complete joined networks of tuples. Totality is enforced by generating CNs that cover all query keywords. For ensuring completeness, Theorem 1 proposed by Hristidis and Papakonstantinou [2002] presents a criterion that determines when the joining networks of tuples produced by a joining network of tuple-sets J has more than one occurrence of a tuple. Thus, to be considered as a candidate network, any joining network of tuple-sets J must avoid this criterion to be fulfilled. In here, we caracterize this property by defining which joining network of tuple-sets are considered as *sound*.

Definition 2.7. We say that a joint network of tuple-sets J is **sound**, if it does not contain a subtree of the form $R^K - S^L - R^M$, where R and S are relations and the schema graph has an edge $R \rightarrow S$.

Intuitively, a CN is a relational join expression that connects subsets of relations from the database whose tuples contain one or more keywords of the query. The "connections" are derived from referential integrity, i.e., PK/FK, constraints, which may involve additional relations. By having a DBMS to evaluate CNs, we obtain semantically meaningful answers as joined tuples which contain the query keywords.

Example 2.1. *As an example, consider the query "*denzel washington gangster*" and suppose we want to execute it over a relational database containing data on movies from the well-known*

Internet Movie Databases (IMDb). A possible CN for this query is given by the relational algebra expression:

$$\sigma_{\text{title}\supseteq\{gangster\}}\,\text{MOV} \bowtie_{\text{id=cid}} \text{CAST} \bowtie_{\text{cid=pid}} \sigma_{\text{name}\supseteq\{denzel,washington\}}\,\text{PER}. \tag{2.1}$$

where MOV *stores information on movies,* PER *stores data on persons (i.e., actors, actresses, directors, etc.) and* CAST *associates persons to movies they they work on. The join conditions in this expression are derived from PK/FK constraints.*

Following the terminology introduced above, the operands of the join operations in a CN are called tuple-sets*. Operands whose tuples contain the keywords specified in the query, such as those defined by selection operations over relations* MOV *and* PER *in Expression 2.1, are called* non-free tuple-sets*. The remaining operands, such as* CAST *in Expression 2.1, are called* free tuple-sets*, since they not contain any of the keywords.*

The complexity of the CN generation is mainly due to two factors: (1) There can be multiple tuple-sets for each subset of terms of the query. As a consequence, there may be a large number of ways of combining these tuple-sets, so that all terms of the query are covered; (2) Given a set of tuple-sets that cover the terms of the query, there can be many distinct ways of connecting them through PK/FK constraints and free tuple-sets.

Chapter 3

Overview of Our Contributions

In this chapter we present an overview of our approaches *MatCNGen* and *CNRank* for generating and ranking Candidate Networks, respectively. For this, we illustrate in Figure 3.1 the steps of a process that begins with a keyword query supplied by an user and ends with the output of some top few Candidate Networks corresponding to the query. In this chapter, each of these steps are introduced and then full descriptions of them are presented in subsequent chapters. Notice that the problem of evaluating Candidate Networks is a out of the scope of this thesis, and we do not discuss methods for dealing with it.

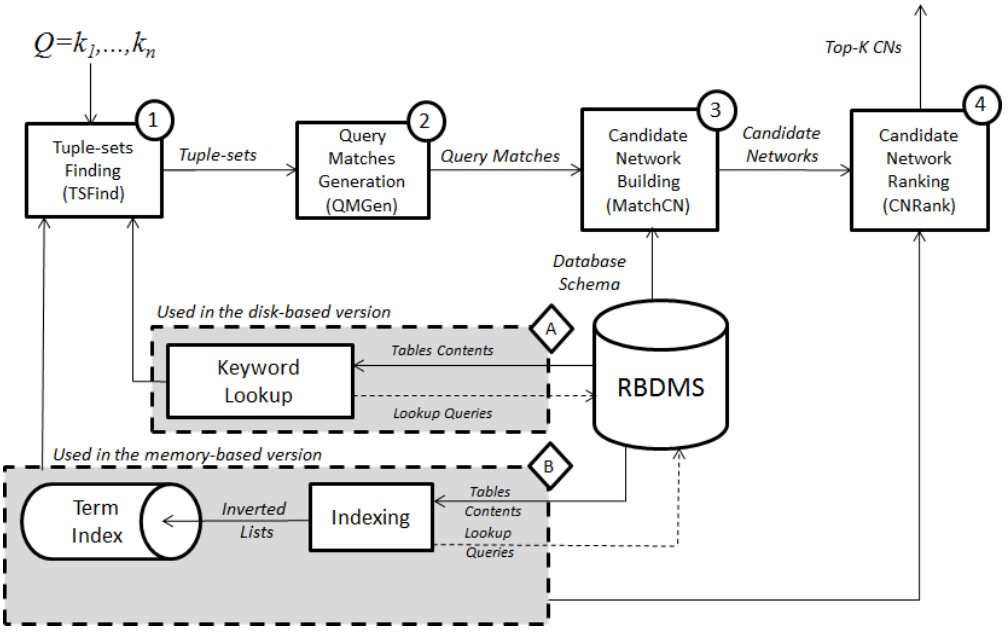


FIGURE 3.1: Overview of the steps for generating and ranking Candidate Networks.

3.1 Tuple-sets Finding

In Figure 3.1, when a keyword query is received for processing, the first step is to identify tuple-sets that can potentially be used in Candidate Networks. This corresponds to Step 1. More precisely, let K be any non-empty subset of the query Q, that is, $K \subseteq Q$, $K \neq \emptyset$. We call K a *termset* of Q. For all termsets of Q and for all relations R from the database, we need to determine if there is some subset of R whose tuples contain all terms from the termset and no other term from the query.

Thus, the finding of tuple-sets starts by determining which relations and tuples contain the keywords from the query. In our work, we consider two alternative strategies for this task. In the first strategy, we lookup on database relations for tuples that contain each of the keywords. We then obtain, for each keyword, a list of the tuples containing it. Thus, we perform an access disk for each keyword query entered. This strategy is illustrated in Figure 3.1, in the shaded box labelled with A. The version of *MatCNGen* that uses this strategy is called the *disk-based* version.

In our second strategy, the lists of tuples that contain each keyword are obtained directly from a data structure we call the *Term Index*.The Term Index associates each term k from the query to a list, that is, an inverted list, whose elements are triples of the form $\langle A_i, f_{k,i}, T_{k,i} \rangle$, where A_i identifies an attribute within a relation in whose values k occurs with frequency $f_{k,i}$, and $T_{k,i}$ is the set of IDs of the tuples from this relation in which k occurs as values of A_i. The term index is built in a preprocessing step that scans once all the relations over which queries will be issued. This step precedes the processing of queries and we assume that it does not need to be repeated often. Under this assumption, CNs are generated for each query without further interaction with the DBMS. This second strategy is illustrated in Figure 3.1, in the shaded box labelled with B. The version of *MatCNGen* that uses this strategy is called the *memory-based*, since the term index is stored in memory.

Once the lists of tuples for each keyword is obtained, by one of the strategies, our algorithm for finding tuple-sets computes non-empty intersections of these lists to find all subsets of the relations from the database in which all tuples, if any, contain the terms of some termset. For instance, in Example 2.1 (Section 2), a tuple-set for the termset {*denzel*,*washington*} over relation PER is found, since there are tuples from this relation that contains these two terms, and no other terms from the query. In the case of the tuple-set over relation MOV, it is generated using *gangster* only, since this relation has tuples that do not contain any other keyword from the query but *gangster*.

In our approach, the intersections are computed in memory using an algorithm we developed based on the ECLAT algorithm [Zaki, 2000]. This algorithm, called **TSFind**, mines termsets of increasing size, starting from one, and it is high scalable. The way MatCNGen addresses

the task of finding tuple-sets is a major difference from what is done in all systems based on DISCOVER's CNGen algorithm [Hristidis and Papakonstantinou, 2002]. In DISCOVER, a module called the *Tuple-Set Post-Processor* materializes tuple-sets as relations in the database by executing several INTERSECT commands involving the original relations The first component is the **Tuple-sets Finding** when a keyword query is received for processing, the first step is to identify tuple-sets that can potentially be used in candidate networks. This corresponds to Step 1 in Figure 3.1. More precisely, let K be any non-empty subset of the query Q, that is, $K \subseteq Q$, $K \neq \emptyset$. We call K a *termset* of Q. For all termsets of Q and for all relation R from the database, we need to determine if there is some subset of R whose tuples contain all terms from the termset and no other term from the query.

The details on the **TSFind** algorithm and the two strategies are discussed in Chapter 5.

3.2 Query Matches Generation

Given a CN such as the one defined by Expression 2.1, we call the set of its non-free tuple-sets as *query match*. As we detail latter, in *MatCNGen* we adopt match subgraph approach for obtaining CNs, which first generates query matches and then uses them to build CNs. The generation of query matches corresponds to Step 2 in Figure 3.1.

Intuitively, each query match represents a different way of combining tuple-sets. Assuming that answers must contain all the query terms, it follows that every keyword must appear in at least one tuple-set in a candidate network. Thus, the union of all terms used in the predicates of a query much form a *set cover* of the query.

In our CN example, the query match is given by the following expression:

$$\{ \underset{\text{title}\supseteq\{gangster\}}{\sigma}\text{MOV} \ , \ \underset{\text{name}\supseteq\{denzel,washington\}}{\sigma}\text{PER} \ \} \tag{3.1}$$

In this case, the cover of the input query is $\{\{gangster\},\{denzel,washington\}\}$.

In general, many more combinations of tuple-sets that generate covers of the query may exist in the database. Thus, many matches may exist and many CNs can be built by connecting the tuple-sets from a match. In *MatCNGen*, we use these ideas to separate the generation of CNs in two distinct steps. First, our method combines the mined tuple-sets to form query matches. Then, it processes the database schema, modeled as graph, looking for ways of connecting the tuple-sets in each query match and build CNs. In Chapter 4 we present an algorithm to generate query matches.

Notice that if the same keywords are spread among many tuples and relations, there can be a large number of query matches. For instance, in the CIA Facts database, which we use in our experiments, terms such as "Africa" and "Economy" are very frequent and spread throughout the database. The set of query matches for such a query must cover all these occurrences. However, only combinations of keywords that correspond to set covers are considered. We also notice that by imposing that query matches must contain set covers of the input query, we in fact impose that matches are minimal and complete. Thus, the CNs assembled using them are also minimal and complete, as required in Definition 2.6.

3.3 Candidate Network Building

In Step 3 of Figure 3.1, each query match generated in Step 2 is used to build CNs. The problem here is to "connect" the tuple-sets that form the query matches through one or more free tuple-sets, i.e., relations in the query graph that create paths between the tuple-sets. In *MatCNGen*, this step consists of building, for each query match M_i, a graph called a *match graph*. This graph contains all relations from the original schema graph, plus nodes corresponding to tuples-sets from M_i. These nodes are linked to the original relations according the PK/FK that exist between their base relations and the original relations.

We notice that, according to Hristidis and Papakonstantinou [2002], the CNGen algorithm works by trying to extract CNs as trees from a graph G_{TS} which includes *all* possible tuples-sets connected to the original relations. In our case, CNs are build from smaller graphs, the match graphs, as trees that connect *only* the tuple-set forming this graph. Thus, while CNGen requires exhaustively exploring the full, and sometimes explosive, combination of the all keywords occurrences in the database and the multiple ways these occurrences can be joined, *MatCNGen* requires exploring several smaller graphs with only a few number of keywords occurrences. This is done for each query match at a time, drastically reducing the cost of exploring the full graph G_{TS}. Interestingly, our match graphs can be regarded as subgraph of the graph G_{TS} induced [Diestel, 2012] by query matches.

In Figure 3.1 this step is represented by an algorithm we call **MatchCN**, which builds the match graphs. For carrying out the searching for joins among tuple-sets in each match graph to ultimately obtain Candidate Networks, we propose two alternative algorithms. The first, called **SingleCN**, is based on the well-known breadth-first traversal algorithm. The second, called **SteinerCN**, is based on the concept of Steiner Trees [Dreyfus and Wagner, 1971], which generalize graph connectivity concepts such as shortest-paths and minimal spanning trees. These algorithms are detailed in Chapter 5.

Besides reducing the time for generation CNs, *MatCNGen* has also a positive impact on the evaluation of CNs. Specifically, as the generation process prunes likely spurious query matches, a smaller but better set of CNs is obtained. As a result, state-of-the-art CN evaluation algorithms [Hristidis et al., 2003; Luo et al., 2007a] run faster and produce higher quality results. These trends were observed across many distinct queries and datasets in the experiments we performed and report in Chapter 9.

3.4 Ranking of Candidate Network

In the next step (4), the CNs generated are ranked using the **CNRank** algorithm , which is presented in Chapter 7. While in current systems all CNs generated must be evaluated, we propose that only a few top CNs in the ranking must be evaluated. The experiments we performed and report in Chapter 10 show that our **CNRank** algorithm is highly effective in the task of placing the best CNs, that is, those that fulfill the user's intention, in top positions of the ranking. These experiments also show that the results achieved when evaluating just a few top CNs are at least as good as those obtained with traditional systems that evaluate all CNs. On the other hand, the overall system performance is considerably improved by introducing **CNRank** between the generation and evaluation of CNs. For our experimental evaluation we instantiated this architecture with implementations of well-known algorithms for generating and evaluating CNs. In practice, the evaluation process can be carried out by any of the many algorithms proposed for this purpose in the literature; for instance, DISCOVER [Hristidis and Papakonstantinou, 2002], DBXplorer [Agrawal et al., 2002], Efficient [Hristidis et al., 2003], SPARK [Luo et al., 2007a], CD [Coffman and Weaver, 2010c] and Min-cost [Ding et al., 2007].

Given a set of Candidate Networks generated for a keyword query, we want to assign to each CN a score value that estimates the likelihood of this CN representing the user intention when formulating the query. In our work, the score a CN is computed as the joint probability of the keywords to compose of values of attributes of its tuple-sets considering the current state of the database. The computed CN scores are then used to rank the CNs based on the belief that they correctly represent the keyword query posed by the user. This process is called *Candidate Network Ranking*.

To estimate this joint probability, the individual probabilities involving the keywords and tuples-sets are combined using a Bayesian network model [Ribeiro and Muntz, 1996]. The processes of ranking CNs is supported by the same *Term Index* used in the tuple-sets finding process.

In Chapter 7 we presented details on our CN ranking algorithm.

Chapter 4

Inducing Schema Subgraphs

In this chapter we formalize our strategy to divide a tuple-set graph into subgraphs for the problem of generating CNs, introduce the concepts of query matches and induced schema subgraphs, which are central in our approach, and present our algorithm for efficiently combining tuple-sets to form query matches.

4.1 Query Matches

Definition 4.1. Let Q be a query. A set of tuple-sets $M = \{R_1^{K_1}, \ldots, R_m^{K_m}\}$, where every K_i is a distinct termset of Q, is a **match** for Q. M is called a **total and minimal match** for Q if K_1,...,K_m form a **minimal set cover** for the set of keywords in Q, that is, $K_1 \cup \ldots \cup K_m = Q$ and $(K_1 \cup \ldots \cup K_m) \backslash K_i \neq Q$, for any termset K_i.

Intuitively, a query match is a set of tuple-sets that, if properly joined, can produce networks of tuples that fulfill the query. They can be thought as the leaves of a Candidate Network. In total and minimal matches, to ensure totality, all keywords from the query must occur in at least one tuple-set of the match. Furthermore, to ensure minimality, there can be no superfluous tuple-set, that is, if we remove any tuple-set from the match, it turns to be non-total.

In this thesis we closely follow the semantics proposed by Hristidis and Papakonstantinou [2002] and only address total and minimal matches. Dealing with other types of match is left for future work. From now on, we use the term *match* to refer to total and minimal matches.

Example 4.1. *The following example is based on the sample of the IMDb database made available by Coffman and Weaver [2010b], whose schema graph is presented in Figure 4.1*[1].

[1] Names of relation and attributes were changed for convenience

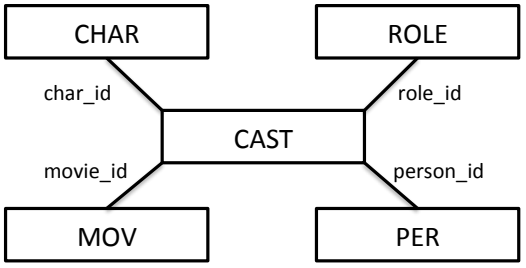


FIGURE 4.1: Schema Graph for the IMDb database.

Consider a query Q={denzel,washington,gangster }. For simplicity, we will use Q={d,w,g}. This query has eight minimal covers, among them:

C_1={{d},{w},{g}}

C_2={{d,w},{g}}

C_3={{d,g},{d,w}} *and so on.*

Consider only cover C_2, whose termsets are {d,w} and {g}. If the keyword "gangster" occurs with no other keywords from Q in tuples of relations CHAR, MOV *and* CAST, *the non-empty tuple-sets for this single-keyword termset are* $\text{CHAR}^{\{g\}}$,$\text{MOV}^{\{g\}}$ *and* $\text{CAST}^{\{g\}}$. *Also, if keywords "denzel" and "washington" occur together, but with no other keywords from Q, in tuples of relations* PER *and* CAST, *the following tuple-sets are also non-empty:* $\text{PER}^{\{d,w\}}$, $\text{CAST}^{\{d,w\}}$. *Thus, considering only termsets in C_2, some of the possible matches for Q are:*

M_1={$\text{CHAR}^{\{g\}}$, $\text{PER}^{\{d,w\}}$} M_2={$\text{CHAR}^{\{g\}}$, $\text{CAST}^{\{d,w\}}$}
M_3={$\text{MOV}^{\{g\}}$, $\text{PER}^{\{d,w\}}$} M_4={$\text{MOV}^{\{g\}}$, $\text{CAST}^{\{d,w\}}$}
M_5={$\text{CAST}^{\{g\}}$, $\text{PER}^{\{d,w\}}$} M_6={$\text{CAST}^{\{g\}}$, $\text{CAST}^{\{d,w\}}$}

Considering all minimal covers from Q, there are 19 distinct matches for this query in our sample of the IMDb database. These matches are combinations of 10 distinct non-empty non-free tuple-sets found in this database for Q.

The role of query matches in CNs is formalized below.

Lemma 4.2. *Let C be a joining network of tuple-sets and let $M = \{R_1^{K_1}, \ldots, R_m^{K_m}\}$ be the set of all its non-free tuple-sets. If C is a candidate network for a query Q, then M must be a match for Q.*

Proof. According to Definition 2.6, a CN must generate minimal and total joining networks of tuple-sets that satisfy the query Q. Thus, to ensure totality, every keyword from Q must apear in at least one non-free tuple-set $R_i^{K_i}$, that is, $K_1 \cup \ldots \cup K_m = Q$. In addition, to ensure minimality, $(K_1 \cup \ldots \cup K_m) \backslash K_i \neq Q$, for any K_i, that is, removing any tuple-set $R_i^{K_i}$ from

M makes the set of non-free tuple-sets non-total. Thus, $\{K_1, K_2, \ldots, K_m\}$ must be a minimal cover for Q. Furthermore, if any $R_i^{K_i} = \emptyset$, no MTJNT for Q can be generated, and C cannot be a candidate network. □

4.2 Obtaining Query Matches

Let Q be a query. The set of all possible termsets of Q is given by $\mathscr{P}^+(Q)$[2]. We use the notation $\mathcal{R}(K)$ to refer to the set of all non-empty non-free tuple-sets of the form R^K, where $K \neq \emptyset$, that can be obtained from any relation R from the database for termset K. Also, we use the notation $\mathcal{R}_Q$ to refer to the set of all tuple-sets, for all termsets from Q, that is: $\mathcal{R}_Q = \cup \{\mathcal{R}(K) | K \in \mathscr{P}^+(Q)\}$

By Definition 4.1, every match for a query Q is a set of tuple-sets of the form $\{R_1^{K_1}, \ldots, R_m^{K_m}\}$, where $\{K_1, \ldots, K_m\}$ is a minimal cover for Q. Then, the set of all possible query matches for Q is given by the combinations of all its tuple-sets on the database whose termsets form minimal covers of Q.

$$\mathcal{M}_Q = \{\{R_1^{K_1}, \ldots, R_m^{K_m}\} \in \mathscr{P}^+(\mathcal{R}_Q) \,|\, \{K_1, \ldots, K_m\} \in MC(Q)\} \tag{4.1}$$

where $MC(Q)$ is the set of minimal covers for Q.

At a first glance, the way we state Equation 4.1 may suggest that we need to generate the whole power set of $\mathcal{R}_Q$ to obtain the complete set of query matches. However, it can be shown that any minimal cover of a set of n elements has at most n subsets [Hearne and Wagner, 1973]. This means that no match of a query with n keywords can be formed by more than n tuple-sets.

This property is exploited in the procedure we use to generate query matches. In Figure 4.2, we sketch a high-level description of this procedure. The **QMGen** algorithm takes as input the keyword query Q and the set $\mathcal{R}_Q$ of non-empty non-free tuple-sets obtained from the database Q. Our strategy for obtaining $\mathcal{R}_Q$ is detailed in Chapter 6.

The algorithm generates all subsets of $\mathcal{R}_Q[i]$ with sizes $i = 1, 2, \ldots, |Q|$ (Line 7) of $\mathcal{R}_Q$. Then, in Line 9, the algorithm selects as query matches those subsets of tuple-sets whose keywords form minimal covers of Q.

It is easy to see that the **QMGen** algorithm has a time complexity of

$$\sum_{i=1}^{|Q|} \binom{|\mathcal{R}_Q|}{i} \tag{4.2}$$

[2]The notation $\mathscr{P}^+(X)$ is used here to refer to the power set of set X, minus the empty set.

1: **QMGen**(Q,$\mathcal{R}_Q$)
2: **Input:** A keyword query Q
3: **Input:** The set of non-empty non-free tuple-sets $\mathcal{R}_Q$
4: **Output:** The set $\mathcal{M}_Q$ of query matches for Q
5: $\mathcal{M}_Q \leftarrow \emptyset$
6: **for** $i = 1, \ldots, |Q|$ **do**
7: **let** $\mathcal{R}_Q[i]$ be set of subsets of size i of $\mathcal{R}_Q$
8: **for each** $\{R_1^{K_1},\ldots,R_i^{K_i}\} \in \mathcal{R}_Q[i]$ **do**
9: **if** $\{K_1, \ldots, K_i\} \in MC(Q)$ **then**
10: $\mathcal{M}_Q \leftarrow \mathcal{M}_Q \cup \{\{R_1^{K_1},\ldots,R_i^{K_i}\}\}$
11: **end if**
12: **end for**
13: **end for**
14: **return** $\mathcal{M}_Q$

FIGURE 4.2: Query Matches Generation

That is, its running time depends on the size of the query and on the size of the sets of tuple-sets $\mathcal{R}_Q$. More important, this equation also gives us an upper bound on the number of query matches that must be generated for a query.

Regarding these two factors, the first one, the size of a keyword query is usually small, e.g., less than two on average, and queries with more than four keywords are rare. In such cases, this summation turns to be a low-degree polynomial. The second factor, $|\mathcal{R}_Q|$, is also dependent on the query size, but the main issue to observe is how query termsets are distributed among database relations. This factor is harder to predict, but usually very few subsets of query terms are frequent in many relations. In fact, larger subsets are increasingly less frequent. Thus, in practice, just a few query matches need to be generated.

Example 4.2. *To illustrate how large the set of query matches can be in practice, consider the query $Q' = \{denzel,washington\}$. In the IMDb database made available by Coffman and Weaver [2010b] the set $\mathcal{R}_{Q'}$ has six non-empty tuple-sets. Thus, 21 subsets of $\mathcal{R}_{Q'}$ are generated by the **QMGen** algorithm, and out of which only five turn to be query matches.*

Now, by adding a single term to the query, we have $Q = \{denzel, washington,gangster\}$ and $\mathcal{R}_Q$ has ten non-empty tuple-sets. This leads to 175 subsets of size up to three, out of which only 19 turn to be query matches.

Further insights on the number of query matches that are typically generated and on the running times of the **QMGen** algorithm in practice will be provided when we report our experimental results in Chapter 9.

4.3 Induced Schema Subgraphs

Up to this point, we characterize the set of non-free tuple-sets that a CN must contain by means of the concept of a query match. According to Definition 2.6, the remaining tuple-sets in a CN are free tuple-sets that connect its non-free tuple-sets to form a tree, that is, a JNT (Joint Network of Tuple-sets).

According to Hristidis and Papakonstantinou [2002], the generation of candidate networks is a procedure that extracts JNTs from a graph the represents the possible ways of connecting the tuple-sets of the query. This graph is called *a tuple-set graph*.

Definition 4.3. A **tuple-set graph** G_{TS} for a query Q is a graph whose nodes are all the non-empty tuple-sets $R_i^{K_i}$, where $K_i \subseteq Q$, including the free tuple-sets, and there is an edge $\langle R_i^{K_i}, R_j^{K_j} \rangle$ in G_{TS} if the schema graph G_u has an edge $\langle R_i, R_j \rangle$.

In our case, query matches supply in advance the non-free tuple-sets that must compose the candidate networks. Thus, for deriving the CNs that include a given query match, we may consider only the non-free tuple-sets that form this match and disregard all others. For this, we rely on the concept of *induced subgraphs* to fragment the tuple-set graph G_{TS} into smaller graphs from which we can generate CNs.

Definition 4.4. Let G_{TS} be a tuple-set graph for a query Q and let M be a query match of Q. We define a **match subgraph** of G_{TS} for M as the subgraph $G_{TS}[M \cup F]$ of G_{TS} induced by $M \cup F$. As all match subgraphs of G_{TS} in fact include the set F, we use the shorter notation $G_{TS}[M]$ to refer to $G_{TS}[M \cup F]$.

Definition 4.4 relies on the fundamental concept of subgraphs induced by subsets of vertices [Diestel, 2012] to characterize a match subgraph $G_{TS}[M]$. $G_{TS}[M]$ is, thus, a subgraph of G_{TS} whose nodes are those in M, that is, the non-free tuple-sets composing match M, and in F, the set of free tuple-sets from G_{TS}. Also, its edges are the edges from G_{TS} that have both endpoints in $M \cup F$.

Example 4.3. *Considering the same database and query from Example 4.1, Figure 4.3 shows the full tuple-set graph and Figure 4.4 shows two of its match graphs. These match graphs were induced by matches M_2 and M_3, respectively, which highlighted with shaded nodes. In all these graphs, notice that the free tuple sets correspond to the relations in the schema graph of Figure 4.1.*

Recall from Section 4.2 that, as our example query has three keywords, any query match will have at most three tuple-sets and, thus, any match graph $G_{TS}[M_i]$ will include at most eight nodes, that its, three non-empty non-free tuple-sets plus five free tuple-sets.

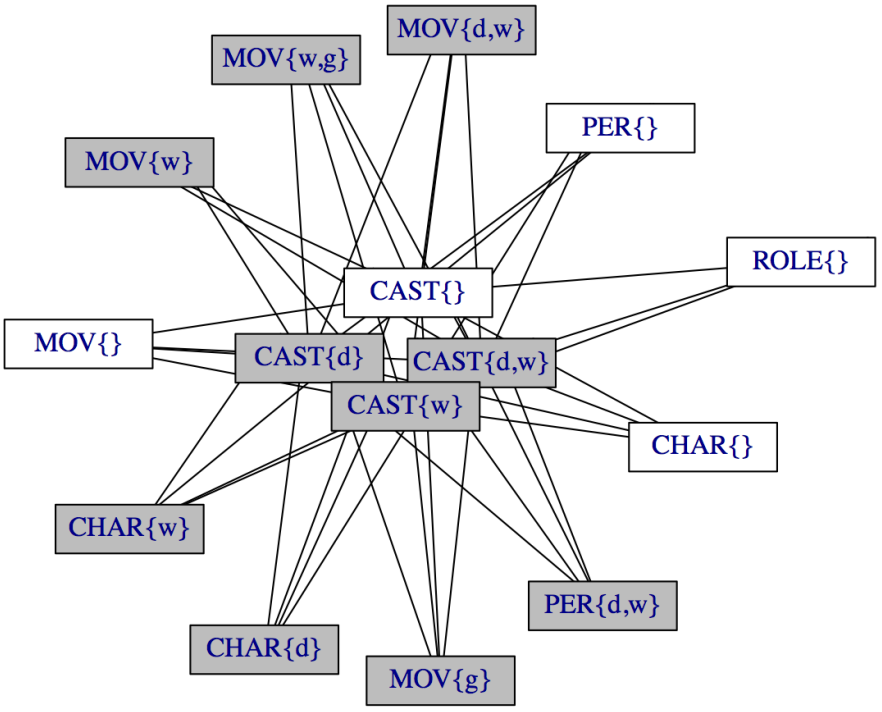


FIGURE 4.3: The tuple-set graph from the query of Example 4.1.

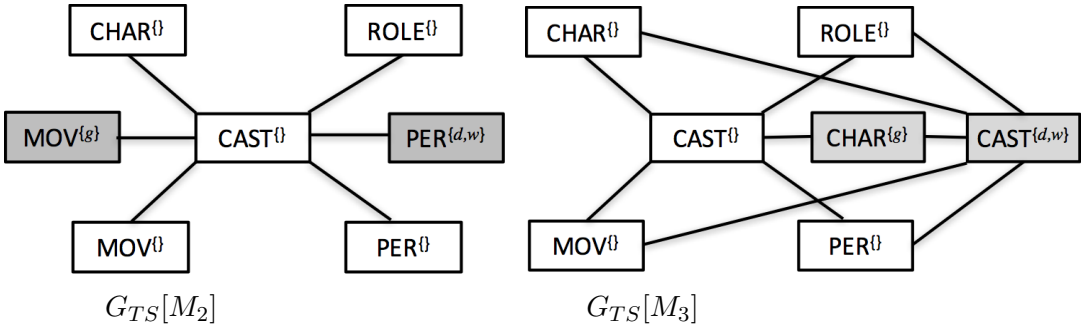


FIGURE 4.4: Two match graphs from the tuple-set graph of Figure 4.3.

Chapter 5

Generation of Candidate Networks

In this chapter we present our algorithms for generating Candidate Networks. We begin by describing the general procedure for generating CNs from match graphs, which we call **MatchCN**. Next, we detail our two alternatives CN generation algorithms, **SingleCN** and **SteinerCN**.

5.1 General Procedure

Figure 5.1 describes the **MatchCN** algorithm, the general procedure we propose to generate CNs from match graphs.

1: **MatchCN**($\mathcal{M}$,G_{TS})
2: **Input:** A set of query matches $\mathcal{M}$
3: **Input:** A tuple-set graph G_{TS}
4: **Output:** A set of CNs $\mathcal{C}$
5: $\mathcal{C} \leftarrow \emptyset$
6: **for each** query match $M \in \mathcal{M}$ **do**
7: **let** $G_{TS}[M]$ be the match graph induced by M from G_{TS}
8: $C \leftarrow$ **CNFind**(M,$G_{TS}[M]$)
9: Add C to $\mathcal{C}$
10: **end for**
11: return $\mathcal{C}$;

FIGURE 5.1: MatchCN Algorithm

The algorithm takes as input a set of query matches $\mathcal{M}$, generated according to the **QMGen** algorithm (Figure 4.2), and a tuple-set graph G_{TS} as in Definition 4.3. In the Loop 6–10, the algorithm process each query match from $\mathcal{M}$ to generate candidate networks corresponding to it. Given a query set M, in Line 7, the algorithm first generates a match graph it induces from the tuple-set graph. Then, Line 8 calls an algorithm to find CNs in this match graph. Finally, the generated CN C is added to set of CNs for the query (Line 9).

Our two alternative algorithms for this task, **SingleCN** and **SteinerCN**, will be detailed in Sections 5.2 and 5.3. Notice that other algorithms for extracting meaningful trees from graphs can be used [Diestel, 2012]. Exploring other algorithms is left for future work.

5.2 The SingleCN Algorithm

The **SingleCN** algorithm is described in Figure 5.2. Given a query match M and a match graph $G_{TS}[M]$, it generates a Candidate Network as the shortest sound JNT that contains the match, if at least one Candidate Networks of size up to T_{max} exists. Thus, notice that at most one CN is generated for each query match. Like the original CNGen algorithm [Hristidis and Papakonstantinou, 2002], **SingleCN** is also based on the well-known breadth-first traversal algorithm.

```
1:  SingleCN(M,G_TS[M])
2:  Input: A query match M, A match graph G_TS[M]
3:  Output: A single CN C
4:  F ← ∅ {Initialize a queue}
5:  J ← {R_1^K1} from M
6:  Enqueue(F,J)
7:  while F not empty do
8:     J ← Dequeue(F)
9:     for each R_v^Kv in G_TS[M] adjacent to some R_u^Ku in J do
10:       if R_v^Kv is a free tuple-set or R_v^Kv ∉ J then
11:          J' ← J
12:          Expand J' with tuple-set R_v^Kv joined to R_u^Ku
13:          if J'∉F and |J'|≤T_max and J' is sound then
14:             if J' contains the match M then
15:                return J' {Return the current JNT as a valid CN}
16:             else
17:                Enqueue(F,J')
18:             end if
19:          end if
20:       end if
21:    end for
22: end while
23: return {}
```

FIGURE 5.2: SingleCN Algorithm

The procedure performs a breath-first traversal over the match graph, starting from the node representing the first tuple-set in the match (Line 4). The Loop 7–22 generates several partial trees, that is, joint networks of tuple-sets (JNT), composed by the tuple-sets in the match and free tuple-sets from the match graph, which are added during the traversal (Line 12).

To be considered to form a valid CN, a JNT J' generated in the loop must satisfy three conditions (Line 13): (1) it must not have been generated previously ($J' \notin \mathcal{F}$) ; (2) its size, i.e., the number

of tuple-sets in it, must not exceed a global threshold T_{max}; and (3) it must be *sound*. Condition (1) ensures that no duplicate CNs will be generated. This was a problem in the original CNGen algorithm, that was latter corrected by Markowetz et al. [2007]. We also avoid this problem here. Condition (2) is considered in the original CNGen algorithm, and prevents the algorithm from generating arbitrarily long CNs. In our experiments we used T_{max}=10. However, as we generate a single shortest CN for each query match, in our experiments we did not found any case in which this condition appeared. The third condition ensures that the joining networks of tuples produced by J' do not have more than one occurrences of a tuple, according to Definition 2.7.

If the current JNT satisfies these conditions, the algorithm verifies if it contains all tuple-sets from the input query match (Line 14). In this case, this JNT is returned as a valid CN (Line 19) and the generation process terminates. Otherwise, the current JNT is enqueued (Line 17) to be further expanded in Line 12.

If during the process none of the JNT generated satisfies the conditions above, the queue will eventually became empty, and the process terminates with no CN being generated (Line 23).

Example 5.1. *Table 5.1 illustrates an execution of the* ***SingleCN*** *algorithm when taking as input the match graph $G_{TS}[M_3]$ of Figure 4.4. Each group of lines in the table refers to an iteration of Loop 7–22, numbered in Column* I#*, except for first line, which corresponds to the initial steps before the loop begins. In this initial step, we consider that $R_1^{K_1} = \mathsf{MOV}^{\{g\}}$, and then an initial JNT with this single tuple-set is generated (Line 5) and enqueued (Line 6). In the first iteration, the current JNT, $\mathsf{MOV}^{\{g\}}$, is expanded with the adjacent tuple-set $\mathsf{CAST}^{\{\}}$, forming a new JNT $\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}}$. As this JNT does not contain the match, it is enqueued. In the next iteration, this JNT is expanded, forming several several new JNTs. Out of these, a single one, $\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}} \bowtie \mathsf{PER}^{\{d,w\}}$, satisfies the conditions of Lines 13 and 14, and is thus returned as valid CN.*

I#	Queue	Operations
0	$\mathsf{MOV}^{\{g\}}$	generate (L.5), enqueue (L.6)
1	$\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}}$	expand (L.12), enqueue (L.17)
2	$\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}} \bowtie \mathsf{CHAR}^{\{\}}$	expand (L.12), enqueue (L.17)
	$\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}} \bowtie \mathsf{ROLE}^{\{\}}$	expand (L.12), enqueue (L.17)
	$\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}} \bowtie \mathsf{MOV}^{\{\}}$	expand (L.12), enqueue (L.17)
	$\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}} \bowtie \mathsf{PER}^{\{\}}$	expand (L.12), enqueue (L.17)
	$\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}} \bowtie \mathsf{PER}^{\{d,w\}}$	expand (L.12), return CN (L.19)

TABLE 5.1: Example of an execution of he **SingleCN** algorithm over the match graph $G_{TS}[M_2]$ of Figure 4.4.

5.3 The SteinerCN Algorithm

In this section we present the **SteinerCN** algorithm, which relies on the concept of *Steiner Trees* [Dreyfus and Wagner, 1971]. Steiner Trees generalize well-known concepts of graph connectivity such as shortest-paths and spanning trees. Although they have been extensively used with R-KwS systems based on data graphs [Aditya et al., 2002; He et al., 2007; Kacholia et al., 2005; Liu et al., 2006], our algorithm is the first one that applies this concept to the context of Candidate Networks. This is only possible due to the use of query matches, a new concept we introduce in our work.

In the following we review the general concept of Steiner Trees, describe our strategy based on Steiner trees to the problem of generating CNs, and present our algorithm based on this strategy.

5.3.1 Concepts

Given an undirected graph $G = \langle V, E \rangle$ and a subset $T \subset V$, a *Steiner Tree* is any tree S that is a subgraph of G and contains all vertices from T, which is called the set of *Terminal Nodes*. If the edges in E have non-negative weights, a *Minimal Steiner Tree* is a Steiner tree with the minimal weight sum among all Steiner trees.

In the following, formalize the problem of finding Candidate Networks as an instance of the Steiner tree problem.

Theorem 5.1. *Let $G_{TS}[M]$ be a match subgraph. Let $\mathcal{S}$ be the set of Steiner trees in $G_{TS}[M]$ whose set of terminals is M. It follows that: (1) every tree in S is a joint network of tuple-sets; (2) every sound joint network of tuple-sets in $\mathcal{S}$ is a Candidate Network.*

Proof. According to Definition 4.4, $G_{TS}[M]$ is composed of tuple-sets of a query Q, where the only non-free tuple-sets are those in M. Thus, any tree extracted from $G_{TS}[M]$ that contains M is a joint network of tuple-sets for Q. Since M is a match for Q, these joint network of tuple-sets are total and complete. If they are sound, then they are Candidate Networks. □

5.3.2 Minimum Steiner Trees

Based on Theorem 5.1, by finding the Steiner trees of a match subgraph and verifying their soundness, we could derive Candidate Networks from it. However, instead of generating all Steiner trees, we chose to generate just the subset of minimum Steiner trees. The rationale is that the minimum Steiner tress are shorter and represent Candidate Networks in which tuple-sets are more tightly connected. Thus, they are more likely to represent the original keyword query.

Notice that $G_{TS}[M]$ is originally an un-weighted graph. Thus, to characterize minimality we need to add weights to it.

Definition 5.2. Let $G_{TS}[M]$ be a match subgraph. Let $\langle R_i^{K_i}, R_j^{K_j} \rangle$ be an edge in $G_{TS}[M]$. We define an weight function c as follows. If both $R_i^{K_i}$ and $R_j^{K_j}$ are free tuple-sets, then $c(\langle R_i^{K_i}, R_j^{K_j} \rangle) = 2$; if any $R_i^{K_i}$ or $R_j^{K_j}$, but not both, are non-free tuple-sets, then $c(\langle R_i^{K_i}, R_j^{K_j} \rangle) = 1$; if both $R_i^{K_i}$ an $R_j^{K_j}$ are non-free tuple-sets, then $c(\langle R_i^{K_i}, R_j^{K_j} \rangle) = 0$;

In Figure 5.3, we presents an example of a match subgraph with weights.

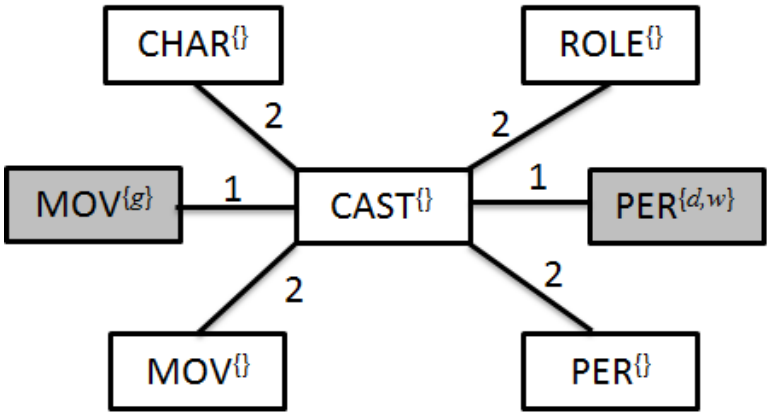


FIGURE 5.3: Match Subgraph with weighted edges.

The problem of minimal Steiner tree problem is known to be NP-Hard in the general case [Dreyfus and Wagner, 1971]. However, two special cases are solvable by efficient algorithms. If all nodes are terminals, then we have the minimum spanning tree problem, and an optimal solution can be found using well known algorithms Kruskal or Prim. If there are exactly two terminal nodes, then we have the shortest-path problem, and an optimal solution can be found using Dijkstra's algorithm. The algorithm we use for the general case is based on an approximate algorithm proposed by Takahashi and Matsuyama [1980].

5.3.3 Algorithm

We describe the **SteinerCN** algorithm Figure 5.4. It takes as input a query match M and a match subgraph $G_{TS}[M]$. The algorithm starts from a tree $T_1 = \langle V_1, E_1 \rangle$ (Line: 6), consisting of a single vertex $R_1^{K_1}$, which is a non-free tuple set from M. In the Loop 7, the algorithm iterates over M to find a vertex $R_j^{K_j}$ that is not included in the current tree, that is, belongs to $M - V_{i-1}$, and that is reached with the minimal possible cost from the current tree. That cost is given by $\hat{c}(V_{i-1}, R_j^{K_j})$, which corresponds to the Dijkstra's algorithm for computing shortest

paths. The path from the current tree to this node is added to for the next tree. The algorithm finishes when all nodes corresponding to the query match are processed. If the resulting tree is sound, according to Definition 2.7, the final tree is returned as a Candidate Network.

1: **SteinerCN**(M,$G_{TS}[M]$)
2: **Input:** A query match M, A match subgraph $G_{TS}[M]$
3: **Output:** T_n
4: $n \leftarrow |M|$
5: **let** $M = \{R_1^{K_1}, \ldots, R_n^{K_n}\}$
6: $V_1 \leftarrow \{R_1^{K_1}\}$; $E_1 \leftarrow \emptyset$; $T_1 \leftarrow \langle V_1, E_1 \rangle$
7: **for each** $i = 2, 3, \ldots, n$ **do**
8: **let** $c_{min} = min\ \{\ \hat{c}(V_{i-1}, R_j^{K_j})\ |\ R_j^{K_j} \in M - V_{i-1}\ \}$
9: **let** $R_j^{K_j} \in M - V_{i-1}$, where $\hat{c}(V_{i-1}, R_j^{K_j}) = c_{min}$
10: $V_i \leftarrow V_{i-1} \cup$ vertices in $minPATH(V_{i-1}, R_j^{K_j})$
11: $E_i \leftarrow E_{i-1} \cup$ edges in $minPATH(V_{i-1}, R_j^{K_j})$
12: $T_i \leftarrow \langle V_i, E_i \rangle$
13: **end for**
14: **if** T_n is *sound* **then**
15: **return** T_n
16: **else**
17: **return** $\emptyset$
18: **end if**

FIGURE 5.4: SteinerCN Algorithm

Example 5.2. *Considering the weigthed match graph $G_{TS}[M_2]$ of Figure 4.4, the **SteinerCN** algorithm starts with $T_1 = \langle \mathsf{MOV}^{\{g\}}, \emptyset \rangle$. In the next iteration, the algorithm selects the other non-free tuple-set from M $\mathsf{PER}^{\{d,w\}}$. The tree $\{\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}} \bowtie \mathsf{PER}^{\{d,w\}}\}$ is returned as a CN.*

5.4 Comparison between MatchCN and CNGen

As we have already noted, **MatchCN** and the original CN generation algorithm, CNGen, are both based on graph traversal procedures. However, the algorithms have important differences to be highlighted.

First, CNGen handles a single graph, the tuple-set graph G_{TS}, that contains all possible non-empty non-free tuples from the query at once, while **MatchCN** handles several smaller graphs, the match graphs. By doing so, **MatchCN** avoids exploring non-relevant paths in the tuple-set graph. For instance, considering our running exemple, in the tuple-set graph for the query Q, illustrated in Figure 4.3, the node corresponding to $\mathsf{CAST}^{\{\}}$ is adjacent to every tuple-set from the other tables in the graph, that is, it has 11 adjacent nodes. The same is true for all other tuple-sets based on relation CAST. Thus there are several redundant and unproductive paths in

the graph that are likely to be explored by CNGen. In **MatchCN**, although there are several small graphs to be processed, these graphs are much simpler, and are free from several non-relevant paths.

Second, in order to generate several alternative CNs, CNGen must be exhaustive and cannot stop until all possible paths have grown up to a threshold (T_{max}). This is the main cause of the excessive resources consumption reported in the literature. In **MatchCN**, using **SingleCN** each graph traversal finishes as soon as a valid CN is found or using **SteinerCN** generates minimal Steiner tree with constraints to produces a set of valid CNs. As this is done for every query match, there will be one CN representing each possible way of distributing the keywords among the relations of the database. Thus, **MatchCN** is scalable, as show the results of experiments we report in this thesis.

Chapter 6

Efficient Finding of Tuple-sets

6.1 Motivation

As shown in Figure 4.2, a necessary step for generating query matches is to find the non-free, non-empty tuple-sets corresponding to the input query, that is, the tuple-sets that effectively contain some termset of the query. In this chapter, we present our approach to accomplish this task. We begin by presenting a simple example of what must be done.

Example 6.1. *Figure 6.4(b) shows the tuple-sets that would be generated for the query* $Q = \{$*denzel, washington, gangster*$\}$*, again represented here as* $Q = \{d,w,g\}$*, from a simple database instance presented in Figure 6.4(a). In this example, tuple IDs are composed of a single letter representing a table name followed by single digit representing the tuple number. For instance, the singleton termset* $\{w\}$ *occurs in tuples* 4 *of table* CAST *and* 2 *of table* PERSON.

Table	Tuple	Contains
CAST	$C1$	*g*
CAST	$C2$	*g*
CAST	$C3$	*d,w,g*
CAST	$C4$	*w*
PERSON	$P1$	*d*
PERSON	$P2$	*w*
PERSON	$P3$	*d,w*

(a)

Tuple-sets	Tuples
$\text{CAST}^{\{g\}}$	$C1, C2$
$\text{CAST}^{\{w\}}$	$C4$
$\text{CAST}^{\{d,w,g\}}$	$C3$
$\text{PERSON}^{\{d\}}$	$P1$
$\text{PERSON}^{\{w\}}$	$P2$
$\text{PERSON}^{\{d,w\}}$	$P3$

(b)

FIGURE 6.1: Examples of tuple-sets (b) from a database instance (a).

Notice that from table PERSON *we derive the non-empty, non-free tuple-sets* $\text{PERSON}^{\{d\}}$, $\text{PERSON}^{\{w\}}$*, as well as* $\text{PERSON}^{\{d,w\}}$*, since there are tuples where terms "d" and "w" occur alone (i.e.,* $P1$ *and* $P2$*, respectively) and tuples were they occur together (i.e.,* $P3$*). On the other hand, the non-free tuple-set* $\text{CAST}^{\{d\}}$ *is empty, since no tuple of* CAST *includes "d" without "g" and "w". Thus, this tuple-set is not considered and not listed in the example.*

A straightforward approach to obtain tuple-sets is to scan every table from the database and look for the occurrence of every termset from the input query. This would require looking for all possible termsets of the query in each tuple in the database. To avoid this, DISCOVER [Hristidis and Papakonstantinou, 2002] first scans each table in the database to obtain initial tuple-sets for each individual keyword in them (i.e., singleton termsets), and stores the result in temporary tables in the database. Then, the system issues several SQL queries over these tables to find tuple-sets for termsets composed of two or more keywords. This is done for every input query.

In our work we adopt a different strategy and propose an algorithm that only accesses the database to obtain the initial tuple-sets for singleton termset. These initial tuple-sets are stored in memory and from this point on, our algorithm makes no further disk accesses for the current query. Instead, it progressively computes set intersections in memory to obtain tuple-sets for termsets of the query that have more than one keyword.

To compute intersections efficiently, we frame the problem of finding non-empty, non-free tuple-sets as a problem of finding frequent item sets, where items are keywords, item sets are termsets (i.e., subsets of the input query), "baskets" or transactions are tuple-sets and the support (minimum frequency) needed is 1. Thus, it reduces to the problem of finding tuples where distinct termsets appear at least once. Our algorithm is described next.

6.2 TSFind Algorithm

We propose the **TSFind** algorithm (Figure 6.2), which is based on the ECLAT [Zaki, 2000] algorithm for finding frequent itemsets. Following the notation introduced in Chapter 2, our algorithm aims at building the set $\mathcal{R}_Q$ of existing tuple-sets for a given keyword query Q in a database instance.

The algorithm uses a data structure $\mathcal{P}$ to keep track of pairs of the form $\langle K, T_K \rangle$, where K is a termset and T_K is a set of tuples from database containing K. T_K is used to compute intersections that will form the lists of tuples that contain termsets.

This algorithm has three partes. In the first part, the algorithm finds sets of tuples that contain each keyword of the query. For this, in the Loop 7–14 the algorithm iterates over each keyword k_i in the query and issues a SQL query over each relation R (Loop 9–13) looking for tuples that contain k_i. In our implementation using PostgresSQL, this query uses the operator `ILIKE` over each text attributes of R. This loop builds, for each keyword k_i, the set of tuples which contain k_i (Line 12). When the loop finishes (Line 14), all pairs in $\mathcal{P}$ refer only to singleton termsets. In the second part, the algorithm invokes in Line 16 a recursive procedure called **TSInter** to find sets of tuples containing larger termsets. This procedure will be detailed next. Finally, in the third part, the algorithm adds to the set of non-empty and non-free tuple-sets $\mathcal{R}_Q$ all tuple-sets

1: **TSFind** (Q)
2: **Input:** A keyword query Q=$\{k_1, k_2, \ldots, k_m\}$
3: **Output:** Set of non-free and non-empty tuple-sets $\mathcal{R}_Q$
4: {**Part 1:** Find sets of tuples containing each keyword}
5: **let** D the current instance of the target DB
6: $\mathcal{P} \leftarrow \emptyset$
7: **for each** keyword $k_i \in Q$ **do**
8: **add** $\langle\{k_i\}, \emptyset\rangle$ to $\mathcal{P}$
9: **for each** relation $R \in D$ **do**
10: {Issue queries over R looking for tuples containing k_i}
11: $T \leftarrow$ set of tuples in R containing k_i
12: **update** $\langle\{k_i\}, T_{k_i} \cup T\rangle$ in $\mathcal{P}$
13: **end for**
14: **end for**
15: {**Part 2:** Find sets of tuples containing larger termsets}
16: $\mathcal{P} \leftarrow$ **TSInter**($\mathcal{P}$);
17: {**Part 3:** Build tuple-sets}
18: $\mathcal{R}_Q \leftarrow \emptyset$
19: **for each** $\langle K, T_K\rangle \in \mathcal{P}$ **do**
20: $\mathcal{R}_Q \leftarrow \mathcal{R}_Q \cup \{R^{\{K\}} \mid$ there is some tuple of R in $T_K\}$
21: **end for**
22: **return** $\mathcal{R}_Q$

FIGURE 6.2: TSFind Algorithm

of the form $R^{\{K\}}$ such that there is at least one tuple from relation R in the set T_K of the tuples that contain the term-set K and no other keyword from the query.

The recursive algorithm **TSInter** is presented in Figure 6.3. It takes as parameters a set $\mathcal{P}$ of pairs $\langle K, T_K\rangle$ and uses two auxiliary structures $\mathcal{P}_{cur}$ and $\mathcal{P}_{prev}$, which are similar to $\mathcal{P}$. We describe the algorithm with the help of an example presented in Figure 6.4, which is based on Example 6.1. In this figure, each box represents a pair $\langle K, T_K\rangle$, with K in the top of the box and T_K in the bottom. The figure illustrates $\mathcal{P}$, $\mathcal{P}_{cur}$ and $\mathcal{P}_{prev}$ for two calls of **TSInter**.

In the algorithm, the Loop 5–14 combines pairs of termsets in $\mathcal{P}$. The termsets are numbered and processed in the numbered order, only to avoid processing the same pair of tuplesets twice. In Line 7, a new termset X is constructed as the union of termsets K_i and K_j and the set of tuples T_X that contain X is given by the intersection of the tuples in T_{K_i} and T_{K_j}. The new pair $\langle X, T_X\rangle$ is stored in $\mathcal{P}_{cur}$ (Line 10). Tuples from T_X must be removed from T_{K_i} and T_{K_j}, since tuple-sets that contain T_{K_i} or T_{K_j} cannot contain other keywords from the query. The algorithm makes the necessary updates in the structure $\mathcal{P}_{prev}$ (Lines 11 and 12), whose role is to keep track of the new state of the input $\mathcal{P}$. In our example of Figure 6.4, the first call of **TSInter** proceeds the intersection of termsets with one keyword to find sets of tuples containing termsets with two keywords. For instance, when processing $\langle\{d\}, \{C3, P1, P3\}\rangle$ and $\langle\{w\}, \{C3, C4, P2, P3\}\rangle$ from $\mathcal{P}$, this call adds $\langle\{d, w\}, \{C3, P3\}\rangle$ to $\mathcal{P}_{cur}$ and updates

1: **TSInter**($\mathcal{P}$)
2: **Input:** A set $\mathcal{P}$ of pairs $\{\langle K_1, T_{K_1}\rangle, \ldots, \langle K_n, T_{K_n}\rangle\}$
3: $\mathcal{P}_{prev} \leftarrow \mathcal{P}$;
4: $\mathcal{P}_{curr} \leftarrow \{\}$;
5: **for** $i = 1$ to $n-1$ **do**
6: **for** $j = i + 1$ to n **do**
7: $X \leftarrow K_i \cup K_j$
8: $T_X \leftarrow T_{K_i} \cap T_{K_j}$
9: **if** $T_X \neq \emptyset$ **then**
10: **add** $\langle X, T_X\rangle$ to $\mathcal{P}_{curr}$
11: **update** $\langle K_i, T_{K_i} - T_X\rangle$ in $\mathcal{P}_{prev}$
12: **update** $\langle K_j, T_{K_j} - T_X\rangle$ in $\mathcal{P}_{prev}$
13: **end if**
14: **end for**
15: **end for**
16: **if** $\mathcal{P}_{cur} \neq \emptyset$ **then**
17: $\mathcal{P}_{curr} \leftarrow$ **TSInter**($\mathcal{P}_{curr}$)
18: **end if**
19: **return** $\mathcal{P}_{prev} \cup \mathcal{P}_{curr}$

FIGURE 6.3: TSInter Algorithm

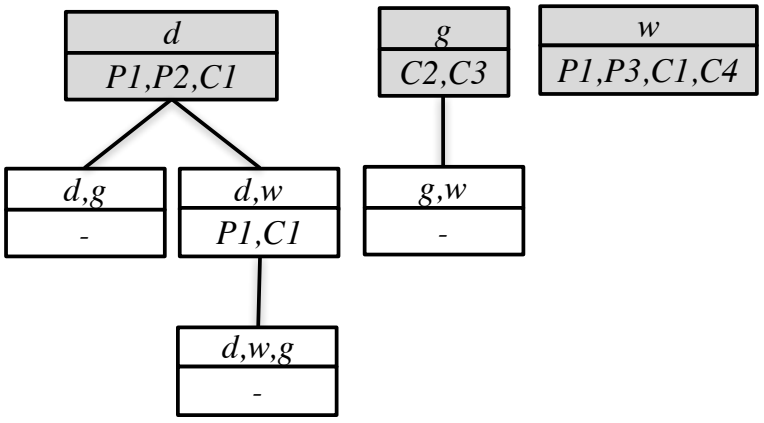


FIGURE 6.4: Finding non-free, non-empty tuple-sets.

$\mathcal{P}_{prev}$ to reflect $\langle\{d\}, \{P1\}\rangle$ and $\langle\{w\}, \{C4, P2\}\rangle$. Similarly, the second call finds sets of tuples containing termsets with three keywords. The algorithm makes a recursive call in Line17 to look for sets of tuples that may include larger termsets. Each call of **TSInter** returns the new state of $\mathcal{P}$, which represented by $\mathcal{P}_{prev}$, expanded with sets of tuples containing larger termsets, which are stored in $\mathcal{P}_{cur}$(Line 19).

6.3 Using an In-Memory Index

As we have discussed, the **TSFind** algorithm is likely to save access operations to the database in comparison with CNGen. However, this algorithm has an initial step (i.e., Part 1) that still requires issuing SQL queries to determine the presence of sought keywords in the tuples of the relations of the database. This is required for each input query.

If the number of expected queries is large enough, a more effective strategy would be scanning all the tables once, building an index based on the terms found in all tuples scanned. Once this index is built, it is possible to generate tuple-sets without further accessing the database for each input query. This index, called *Term Index*, associates each distinct term k with a list of unique tuple IDs where k occurs for an attribute. Effectively, this is an inverted index that gives, for each term, the set of tuples in which this term occurs. It is kept in memory while processing the queries.

The **TSFind_Mem** algorithm, which finds tuple-sets using this index, is described in Figure 6.5. It is very similar to **TSFind**. In fact, the only diference from **TSFind** is on the first part, where the lists of tuples containing the keywords of the query are directly obtaining from the Term Index I, which was previously generated in a pre-processing phase.

1: **TSFind_Mem**(Q)
2: **Input:** A keyword query Q=$\{k_1, k_2, \ldots, k_m\}$
3: **Output:** Set of non-empty and non-free tuple-sets $\mathcal{R}_Q$
4: {**Part 1:** Find sets of tuples containing each keyword}
5: $\mathcal{P} \leftarrow \emptyset$
6: **let** I be a term index for the target DB
7: **for each** keyword $k_i \in Q$ **do**
8: **retrive** from I the list $I[k_i]$ of the tuples containing k_i
9: **add** $\langle \{k_i\}, I[k_i] \rangle$ to $\mathcal{P}$
10: **end for**
11: {**Part 2:** Find sets of tuples containing larger termsets}
12: $\mathcal{P} \leftarrow$ **TSInter**($\mathcal{P}$);
13: {**Part 3:** Build tuple-sets}
14: $\mathcal{R}_Q \leftarrow \emptyset$
15: **for each** $\langle K, T_K \rangle \in \mathcal{P}$ **do**
16: $\mathcal{R}_Q \leftarrow \mathcal{R}_Q \cup \{R^{\{K\}} \mid$ there is some tuple of R in $T_K\}$
17: **end for**
18: **return** $\mathcal{R}_Q$

FIGURE 6.5: TSFind Algorithm – Memory Version

It is easy to conclude that this index-based version of the **TSFind** algorithm improves the time required to handle each individual input query. This was verified in the experiments we carried out and the results are reported in this thesis. However, some questions arise when adopting this alternative. Regarding the time spent for building the index, it is worth noticing that this

task consists of full scanning operations over the tables in the database, which are likely to be performed sequentially on the disk. Thus, this time is expected to be fairly reasonable, considering that these operations are performed once for all queries. For instance, for the largest dataset we experimented with, building the term index took less than 40 seconds.

Also, as the term index is kept in the memory, there could be some concern with exhausting the memory capacity with the index. However, this potential problem is mitigated by the fact that only tuple IDs are stored in the memory, instead of actual tuple contents. Furthermore, in practice, it is possible to avoid indexing unimportant terms such as stop words, therefore contributing to reductions in space requirements. Nevertheless, there are many alternatives to deal with this problem, such as index compression or mechanisms to partially store the term index in secondary memory. Exploring these alternatives is suggested for future work.

Another possible drawback of using the in-memory term index is that updates to the tables would not be reflected in the index, unless it is rebuilt or updated from time to time. Alternatives to this would be running triggers to catch updates to the database and reflecting them in the index in a timely fashion. Again, this problem can be addressed in future work.

Chapter 7

Ranking Candidate Networks

Given a set $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ of Candidate Networks generated for a keyword query U, we want to assign to each CN a score value that estimates the likelihood of this CN representing the user intention when formulating U.

Consider a CN $C = \langle \mathcal{R}, \mathcal{E} \rangle$, where $\mathcal{R}$=$\{R_1^{K_1}, \ldots, R_n^{K_n}\}$ (n>0) is the set of its tuple-sets and E=$\{\langle R_a^{K_a}, R_b^{K_b} \rangle |$ there is a join between $R_a^{K_a}$ and $R_b^{K_b}$ in $C\}$ ($1 \leq a, b \leq n$ and $a \neq b$).

In our work, the score of C is computed as the joint probability of the keywords in each K_j that composse the values of some attribute of R_j, considering the current state of the database. The computed CN scores are then used to rank the CNs based on the belief that they correctly represent the keyword query posed by the user. This process is called *Candidate Network Ranking*.

To estimate this joint probability, the individual probabilities involving the keywords K_j and the relation R_j are combined using a Bayesian network model [Ribeiro and Muntz, 1996] we will present later. We first need to introduce an alternative algebraic representation for Candidate Networks.

Our method for ranking Candidate Networks was first published in a full paper accepted for the IEEE 2015 International Conference on Data Engineering (ICDE) [de Oliveira et al., 2015].

7.1 Algebraic Representation of CNs

According to Hristidis and Papakonstantinou [2002], there are two types of tuple-sets. Let $R_j^{K_j}$ be a tuple-set. If $K_j \neq \{\}$, it is called a *non-free* tuple-set, and is composed of a subset of the tuples from R_j that contain a subset K_j of the keywords in the input query. On the other hand, if $K_j = \{\}$, then we have a *free* tuple-set, and it contains all tuples from relation R_j. Intuitively, the role of *free* tuple-sets in a CN is to "connect" non-free tuple-sets.

Now, consider a CN C_i defined as above. In such a CN, any tuple-set $R_j^{K_j}$ can be represented by a relational algebra expression of the form $\sigma_{\alpha_j}(R_j)$, where α_j is a predicate. The form of α_j depends on the type of the tuple-set. If $K_j \neq \{\}$, that is, if $R_j^{K_j}$ is a non-free tuple-set, then α_j is a predicate of the form $A_j \supseteq K_j$, where A_j is an attribute of R_j. This predicate is true for tuples of R_j in which the value of A_j contains all terms of K_j. Otherwise, if $K_j = \{\}$, that is, if $R_j^{K_j}$ is a free tuple-set, then α_j is a tautology. This means that all tuples from R_j will be included in the tuple-set.

Likewise, we can use an algebraic representation of Candidate Networks using a relational algebra expression of the form:

$$\sigma_{\alpha_1}(R_1) \underset{\Theta_{1,2}}{\bowtie} \sigma_{\alpha_2}(R_2) \underset{\Theta_{1,3}}{\bowtie} \ldots \underset{\Theta_{n-1,n}}{\bowtie} \sigma_{\alpha_n}(R_n)$$

where each $\Theta_{i,i+1}$ $(1 \leq i \leq n-1)$ is a join condition that implements the edge $\langle R_i^{K_i}, R_{i+1}^{K_{i+1}} \rangle$ from the CN.

For convenience, we apply a simple algebraic transformation over the expression above as follows:

$$\underset{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}{\sigma} (R_1 \underset{\Theta_{1,2}}{\bowtie} R_2 \underset{\Theta_{2,3}}{\bowtie} \ldots \underset{\Theta_{n-1,n}}{\bowtie} R_n)$$

Thus, we can represent a CN C as a pair $C = \langle T, P \rangle$, where

$$T = R_1 \underset{\Theta_{1,2}}{\bowtie} R_2 \underset{\Theta_{2,3}}{\bowtie} \ldots \underset{\Theta_{n-1,n}}{\bowtie} R_n \text{ and } P = \alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n$$

where T is called the *base relation* of the CN and P is its set of predicates. Alternatively, this CN can also be represented by $C = \langle T, P' \rangle$, where in P', the tautologies from P are removed.

Example 7.1. *The following example is based on the IMDb database described by Coffman and Weaver [2010b], whose schema graph is presented in Figure 4.1*[1]*. Relations are represented by rectangles labelled with their respective schema definitions. Referential integrity constraints are represented by arcs with keys/foreign keys as labels in arc ends. Consider a keyword query* U = {denzel,washington, gangster}. *For a given database instance, a possible Candidate Network for this query is given by:*

$$\underset{\text{CHAR.name}\supseteq\{\textit{gangster}\} \wedge \text{PER.name}\supseteq\{\textit{denzel,washington}\}}{\sigma} [(\text{CHAR} \underset{\text{id=cid}}{\bowtie} \text{CAST}) \underset{\text{pid=id}}{\bowtie} \text{PER}\,]$$

where the base relation is $\text{CHAR} \underset{\text{id=cid}}{\bowtie} \text{CAST} \underset{\text{pid=id}}{\bowtie} \text{PER}$ *and there are two predicates:*

$\text{CHAR.name}\supseteq\{\textit{gangster}\}$ *and*

[1] Names of relations and attributes were changed for convenience

PER.name$\supseteq\{$*denzel*,*washington*$\}$

7.2 Probabilistic Ranking Model

Given the alternative representation of CNs introduced above, we can describe the Bayesian network model [Ribeiro and Muntz, 1996] we use to compute the score of a Candidate Network. Such a model is illustrated in Figure 7.1. Notice that although the figure represents a single CN, it can be easily expanded to many queries.

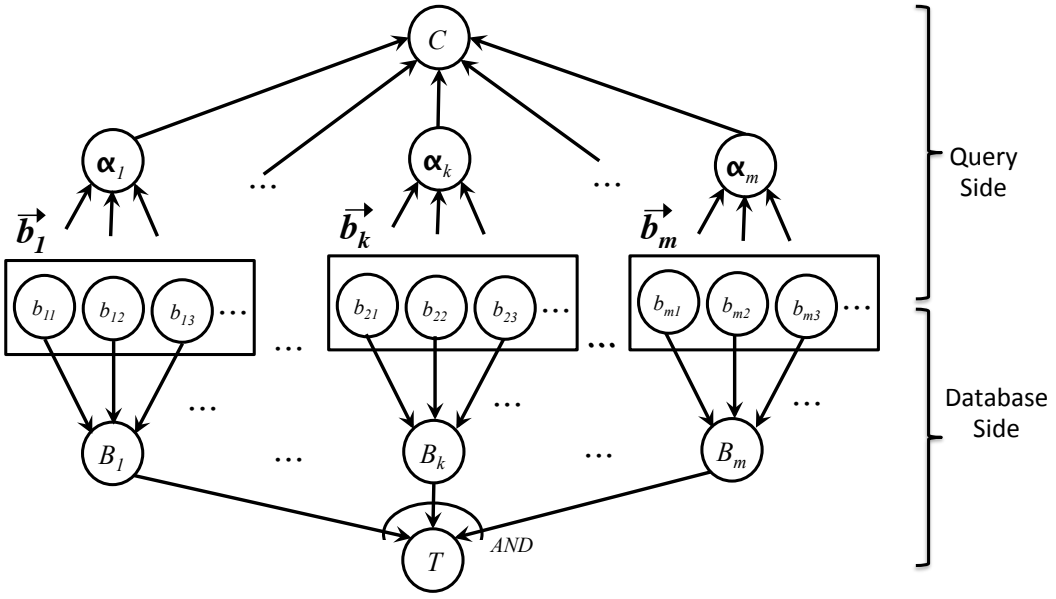


FIGURE 7.1: Bayesian network Model for ranking CNs.

In a Bayesian network, each node represents a piece of information. At the top of Figure 7.1, node C is a Candidate Network and $\alpha_1,\ldots,\alpha_m$ are its predicates. At the bottom, labeled "Database Side", nodes represent the current state of the relations, i.e., the tuple-sets, involved in the CN. Specifically, nodes $B_1,\ldots,B_m$ represent the attributes of these relations and T represents the base relation of C that joins these relations. Each node b_{ij} represents a term found in the values of B_i in the current database state and $\overrightarrow{b_i}$ is a vector containing all terms found in the values of B_i. For simplicity, only attributes that use the predicates are illustrated in this figure.

Example 7.2. *To illustrate these nodes and their roles, in Figure 7.2, we show an instance of the Bayesian network for the CN of Example 7.1. In this case, the Candidate Network side would have only two nodes to represent the two predicates of this CN. The first predicate refers to the term "gangster" found in the values of* CHAR.name *and the second predicate refers to the pair of terms "denzel" and "washington" found in the values of* PER.name.

On the database side, we have the attributes name *from relation* CHAR *and* name *from relation* PER *joined by the base relation of the CN. Then, vector* $\overrightarrow{b_x}$ *will have a node corresponding to*

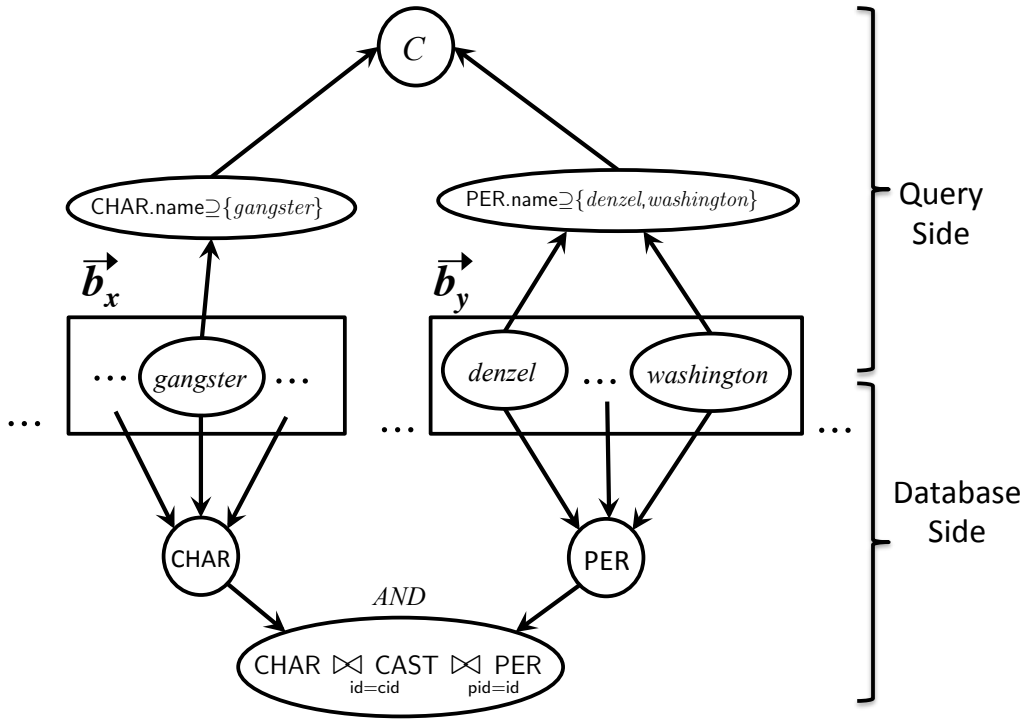


FIGURE 7.2: Example of a Bayesian network for ranking CNs.

term "gangster" and vector $\overrightarrow{b_y}$ will have a node corresponding to each term "denzel" and "washington". We would also have arcs from this term-related nodes to the nodes representing the appropriate predicates.

Each node of the network is associated to a binary variable, which takes the value 1 to indicate that the corresponding information will be considered for the ranking computation. In this case, we say that the information was *observed*. The likelihood of a Candidate Network C, given the current state of the base relation T, can be seen as the probability of observing C, given that relation T was observed. This is represented by conditional probability $P(C|T)$.

Analyzing the Bayesian network in Figure 7.1, we can derive the following general equation:

$$P(C|T) = \mu \times \sum_{\overrightarrow{b_1},\ldots,\overrightarrow{b_m}} \Big(P(C|\overrightarrow{b_1},\ldots,\overrightarrow{b_m}) \times P(T|\overrightarrow{b_1},\ldots,\overrightarrow{b_m}) \times P(\overrightarrow{b_1},\ldots,\overrightarrow{b_m}) \Big) \tag{7.1}$$

where μ is a normalizing constant.

The probability $P(C|\overrightarrow{b_1},\ldots,\overrightarrow{b_m})$ of observing the CN C given the state of all attributes it uses, can be expanded as:

$$P(C|\overrightarrow{b_1},\ldots,\overrightarrow{b_m}) = \sum_{\alpha_1,\ldots,\alpha_m} P(C|\alpha_1,\ldots,\alpha_m)\times P(\alpha_1|\overrightarrow{b_1}) \times \ldots \times P(\alpha_m|\overrightarrow{b_m}) \tag{7.2}$$

In Equation 7.2, the term $P(C|\alpha_1,\ldots,\alpha_m)$ corresponds to the probability of observing C given the predicate nodes α_1 to α_m. As we consider that the CN C should be active only when all its predicate nodes are also active, this translates to the following equation:

$$P(C|\alpha_1,\ldots,\alpha_m) = \begin{cases} 1 & \text{if } \alpha_1 = 1 \wedge \ldots \wedge \alpha_m = 1 \\ 0 & \text{otherwise} \end{cases}$$

Still in Equation 7.2, the terms $P(\alpha_1|\overrightarrow{b_1})$ to $P(\alpha_m|\overrightarrow{b_m})$, which correspond to the probability of observing the nodes α_1 and α_m given the state of the attributes B_1 to B_m, respectively, ensure that the active terms in $\overrightarrow{b_j}$ exactly match the terms in the predicate nodes α_j. This translates to the following equation:

$$P(\alpha_j|\overrightarrow{b_j}) = \begin{cases} 1 & \text{if } \forall k, g_k(\overrightarrow{b_j}) = 1 \text{ iff } t_{jk} \text{ occurs in } K_j \\ 0 & \text{otherwise} \end{cases}$$

where $g_k(\overrightarrow{b_i})$ gives the value of the k-th component $\overrightarrow{b_i}$ and t_{jk} is the kth term in B_j, considering $\alpha_j = B_j \supseteq K_j$.

The sum in Equation 7.1 takes into account all sets of possible active terms in vectors $\overrightarrow{b_1}$ to $\overrightarrow{b_m}$. However, the definition of the probability $P(C|\overrightarrow{b_1},\ldots,\overrightarrow{b_n})$ annuls the probability of any set of active terms, except the set in which the active terms exactly match the query terms referring to attributes B_1 to B_n, respectively. Thus, we can simplify Equation 7.1 to:

$$P(C|T) = \mu \times P(T|\overrightarrow{b_1},\ldots,\overrightarrow{b_m}) \times P(\overrightarrow{b_1},\ldots,\overrightarrow{b_m}) \tag{7.3}$$

where the active terms in vectors $\overrightarrow{b_1}$ to $\overrightarrow{b_m}$ are exactly those present in the predicates of C.

In a similar way, we can expand the probability of observing the base relation T given the state of all its attributes $P(T|\overrightarrow{b_1},\ldots,\overrightarrow{b_m})$ using the probabilities $P(T|B_1,\ldots,B_m)$, $P(B_1|\overrightarrow{b_1})$, $P(B_2|\overrightarrow{b_2})$, etc. Given this and the fact that vectors $\overrightarrow{b_j}$ are independent, Equation 7.1 can be rewritten as:

$$P(C|T) = \alpha \times P(B_1|\overrightarrow{b_1}) \times \ldots \times P(B_m|\overrightarrow{b_m}) \times P(T|B_1,\ldots,B_m) \times P(\overrightarrow{b_1}) \times \ldots \times P(\overrightarrow{b_m}) \tag{7.4}$$

where $\overrightarrow{b_j}$ is the state where only the query terms in C referring to attributes B_j are active.

We can now use the properties of the Candidate Networks to solve the conditional probabilities appearing in Equation 7.4, namely $P(B_j|\overrightarrow{b_j})$, $P(\overrightarrow{b_j})$ and $P(T|B_1, \ldots, B_m)$. Since there is no preference for any particular set of terms, the probability of vector $\overrightarrow{b_j}$ is defined as a constant:

$$P(\overrightarrow{b_j}) = \frac{1}{2^{N_j}}$$

where N_j is the total number of distinct terms that occur for values of the attribute B_j. Notice that N_j is taken from the original table where B_j belongs instead of the base relation.

The remaining probabilities are computed using a model we refer to as TF-IAF, which adapts the traditional vector space model [Baeza-Yates and Ribeiro-Neto, 2011] to the context of relational databases, similarly as done by Mesquita et al. [2007]. In TF-IAF, the cosine measure is used to estimate the probability of observing the attribute B_j in the database, given the terms indicated by $\overrightarrow{b_j}$. The probability of observing B_j is defined as the cosine of the angle between a vector $\overrightarrow{B_j}$ which represents the values of the attribute B_j stored in the database (also considering its original relation), and vector $\overrightarrow{b_j}$ which represents the values for attribute B_j in the query, i.e.,

$$P(B_j|\overrightarrow{b_j}) = cos(\overrightarrow{B_j}, \overrightarrow{b_j}) = \frac{\sum_{t_k \in \overrightarrow{b_j}} w_{jk} \cdot g_k(\overrightarrow{b_j})}{\sqrt{\sum_{t_k \in \overrightarrow{B_j}} w_{jk}^2} \times \sqrt{\sum_{t_k \in \overrightarrow{b_j}} g_k(\overrightarrow{b_j})^2}} \tag{7.5}$$

where w_{jk} is the weight of the term t_k in attribute B_j. To compute the weights, we use the concepts of *term frequency* (*TF*) and *inverse attribute frequency* (*IAF*) [Mesquita et al., 2007].

TF measures the frequency of a given term into the values of an attribute considering all tuples of the relation where it occurs. This is computed by the following formula:

$$tf = \frac{log(1 + f_{kj})}{log(1 + N_j)} \tag{7.6}$$

where f_{kj} is the number of occurrences of term k in the attribute B_j and N_j is the total number of distinct terms that occurs in the values of attribute B_j.

IAF is an adaptation of the concept of inverse document frequency (IDF) found in the context of information retrieval [Baeza-Yates and Ribeiro-Neto, 2011]. Here, it measures how infrequent the term is among the values of the attributes according to the following formula:

$$iaf = log\left(1 + \frac{N_a}{C_k}\right) \tag{7.7}$$

where N_a is the total number of attributes in the database and C_k is the number of attributes in whose values the term k occurs. We use IAF as an estimation of the degree of ambiguity of the term with respect to the attributes in the database.

For instance, consider the relation instance shown in Figure 7.3. In this table, the term *Washington* is considered ambiguous, since it occurs for both attributes Person and Movie. On the other hand, the term *Godzilla* is typical to the Title attribute, and thus it is unambiguous.

Person	Movie
Albert Finney	*Washington Square*
Kerry Washington	*Against the Ropes*
Juliette Binoche	*Godzilla*
Denzel Washington	*Hurricane*

FIGURE 7.3: A sample relation for illustrating the TF-IAF model.

The weights according to the TF-IAF model are computed by the following formula:

$$w_{jk} = tf \times iaf = \frac{log(1 + f_{kj})}{log(1 + N_j)} \times log\left(1 + \frac{N_a}{C_k}\right) \tag{7.8}$$

where f_{kj} is the number of occurrences of the term k in the attribute B_j, N_j is the total number of distinct terms that occur in the values of the attribute B_j, N_a is the total number of attributes in the database and C_k is the number of attributes in whose values the term k occurs.

7.3 Final Ranking Equation

Continuing the expansion of the components in Equation 7.4, since T may be be the result of joining several different relations, we define the probability $P(T|B_1, \ldots, B_m)$ as depending on the number of relations involved in the CN. We would like to prioritize queries in which terms are placed "near" each other, by penalizing queries whose terms are distributed among several relations. We also ensure that the base relations of the CN is observed if all the attributes are observed. Thus, we define this probability as follows:

$$P(T|B_1, \ldots, B_m) = \begin{cases} \frac{1}{|T|} & \text{iff } B_1 = 1 \wedge \ldots \wedge B_m = 1 \\ 0 & \text{otherwise} \end{cases} \tag{7.9}$$

where $|T|$ is the number of relations involved in the CN.

Once all conditional probabilities are defined, we can derive the final equation that represents the probability of observing C given the current state of base relation T. As we use TF-IAF to compute the conditional probability $P(B_j|\overrightarrow{b_j})$, we derive the following equation from Equation 7.4.

$$P(C|T) = \eta \times cos(\overrightarrow{B_1}, \overrightarrow{b_1}) \times \ldots \times cos(\overrightarrow{B_m}, \overrightarrow{b_m}) \times \frac{1}{|T|} \tag{7.10}$$

where each $\overrightarrow{b_j}$ is the state where only the query terms referring to attribute B_j are active, $|T|$ is the number of tuple-sets in the CN and η accounts for the constants μ, and $P(\overrightarrow{b_1})$ to $P(\overrightarrow{b_n})$.

Equation 7.10 is applied to each Candidate Network and returns a probabilistic score for each one.

7.4 Term Index

As mentioned in Section 7.2, to compute Equation 7.10 we use an inverted index that associates each term k found in the relations of the database to a list whose elements are pairs of the form $\langle B_j, f_{kj} \rangle$, where B_j is an attribute in a relation, in whose values the term k occurs with frequency f_{kj}. The term index is constructed in a preprocessing step that scans once all the tables over which queries will be issued. This step precedes the processing of queries, and we assume that it does not need to be repeated often. Under this assumption, CNs are ranked for all queries without further interaction with the DBMS. In experiments we performed with databases used in previous R-KwS papers in the literature, it was possible to store the term index entirely in main memory. Notice that the query index is important to our method since the information it keeps is not usually available in existing database statistics.

7.5 Ranking Algorithm

The **CNRank** algorithm is described in Figure 7.4. It is a direct application of the model we derived in the previous section. The algorithm iterates over a set of CNs given as input (Loop 1–17) and computes a score to each CN according to Equation 7.10. For each CN, the algorithm process each predicate α_j to compute the cosine componentes of this equation (Loop 3–15). In Line 7, the algorithm takes the frequency f_{kj} of term t_k in values of attribute B_j by looking up the entry corresponding to this term in the term index described in Section 7.2 Equation 7.8 in Line 9. Each cosine from Equation 7.10 is computed in Line 13 according to Equation 7.5. Variable $wsum$ is the summation of weights appearing in numerator of Equation 7.5 and the function **anorm**(B_j) computes the term $\sqrt{\sum_{t_k \in \overrightarrow{B_j}} w_{jk}^2}$. The other term, $\sqrt{\sum_{t_k \in \overrightarrow{b_j}} g_k(\overrightarrow{b_j})^2}$ is the same for all CNs of a same query, thus, it does not need to be computed for ranking purposes and can be, thus, omitted. After the scores corresponding to all CNs are calculated, the ranking $\mathcal{R}$ is built (Line 18).

Algorithm **CNRank**
Input: A set of CNs $\mathcal{C} = \{C_1, \ldots, C_n\}$
Output: A rank of CNs $\mathcal{R} = \langle C_{p(1)}, \ldots, C_{p(n)} \rangle$

1: **for each** $C_i \in \mathcal{C}$ **do**
2: $cosprod \leftarrow 1$
3: **for each** $\alpha_j \in C_i$ **do**
4: **Let** $\alpha_j = B_j \supseteq K_j$
5: $wsum \leftarrow 0$
6: **for each** $t_k \in K_j$ **do**
7: $f_{kj} \leftarrow$ TermIndexLookUp(t_k, B_j)
8: **if** $f_{kj} \neq 0$ **then**
9: $w \leftarrow$ tfiaf(f_{kj}, t_k, B_j)
10: $wsum \leftarrow wsum + w$
11: **end if**
12: **end for**
13: $cos \leftarrow wsum/$**anorm**(B_j)
14: $cosprod \leftarrow cosprod \times cos$
15: **end for**
16: $score_i \leftarrow \eta \times cosprod \times |T|$
17: **end for**
18: Build $\mathcal{R}$ such that $C_{p(a)} \preceq C_{p(b)}$, iff $score_a \geq score_b$

FIGURE 7.4: The CNRank algorithm

Chapter 8

Experimental Setup

In this chapter we present the configuration we used to performed a number of experiments we carried out to evaluate the algorithms we proposed in this thesis. The results of the experiments will be presented in the subsequent chapters.

8.1 Hardware

We ran the experiments on an AWS Virtual Machine (medium, 64-bit, 16 GiB RAM, 1ECU, 1vCPU, 160GB of Instance Storage, low Network performance) running on Ubuntu Linux. We used PostgreSQL as the underlying RDBMS with a default configuration. All implementations were made in Java.

8.2 Baselines

Our experiments used representative state-of-art systems described in the literature for processing keyword queries over relational databases. This includes systems based on the Candidate Network approach, like DISCOVER [Hristidis and Papakonstantinou, 2002], Efficient [Hristidis et al., 2003], Bi-directional [Kacholia et al., 2005], Effective [Liu et al., 2006], SPARK [Luo et al., 2007a] and CD [Coffman and Weaver, 2010c], as well as systems based on the Data Graph approach, such as BANKS [Aditya et al., 2002], Min-cost [Ding et al., 2007] and BLINKS [He et al., 2007].

8.3 Datasets

We used five datasets: IMDb, Mondial, Wikipedia, DBLP and TPC-H, which were previously used for the experiments reported by Coffman and Weaver [2010a], Luo et al. [2007a] and others previous work. In Table 8.1 we present some details on these datasets, including their size (in MB), the number of relations, the total number of tuples and the number of Referential Integrity Constrains (RIC) in their schemas. Further details on IMDb, Mondial and Wikipedia datasets are provided by Coffman and Weaver [2010b], whereas more details on the DBLP dataset aregiven by Cyganiak [2007]. We use the TPC-H database with size of 876MB instead of 100MB as done by [Hristidis and Papakonstantinou, 2002].

Dataset	Size (MB)	Relations	Tuples	RIC
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

TABLE 8.1: Characteristics of the databases used.

8.4 Query Sets

In an effort to provide a fair comparison with previous work, we used three query sets from different sources as summarized below.

Coffman-Weaver

Queries used in the experiments reported by Coffman and Weaver [2010b]. There were 42 to 45 queries targeted to the IMDb, Mondial and Wikipedia.

SPARK

Queries used in the experiments reported by Luo et al. [2007b]. For the IMDb dataset, seven out of the 22 queries originally proposed were replaced. This decision was made because these queries refer to a table on film genres, and this table was not available in the sample of IMDb obtained from Coffman and Weaver [2010b]. We then replaced these queries with equivalent queries that mention person names instead. The remaining 15 queries were used without modification. For the DBLP dataset, two out of 18 queries were replaced because they do not have results in the database. For the Mondial dataset, all 35 original queries were used.

INEX

Fourteen queries originally specified for the INEX 2011 challenge [INEX, 2011] that could be applied for searching in relational databases. The other 29 queries from the challenge were disregarded in our experiments, because they mentioned structural XML elements.

In total, we experimented with a total of 218 queries. An overview of our experimental query sets ins presented in Table 8.2.

Dataset	Number of Queries			
	Coffman-Weaver	SPARK	INEX	Total
IMDb	42	22	14	78
Mondial	42	35	-	77
Wikipedia	45	-	-	45
DBLP	-	18	-	18
TOTAL	129	75	14	218

TABLE 8.2: Overview of our experimental query sets.

An important parameter that impacts CN generation is the number of keywords used in the queries. In Table 8.3 we present the maximum and the average number of keywords used in the queries of the query sets we experiment with. Considering all query sets, the average number of keywords per query is 2.1 and the maximum is four. These number are indeed typical of keyword queries posed by users in general.

Dataset	Number of Keywords					
	Coffman-Weaver		SPARK		INEX	
	Max	Avg	Max	Avg	Max	Avg
IMDb	4.00	2.00	3.00	2.31	4.00	2.42
Mondial	3.00	1.40	3.00	2.25	-	-
Wikipedia	3.00	1.91	-		-	-
DBLP	-	-	4.00	2.68	-	-

TABLE 8.3: Max and Avg number of keywords in the queries.

Notice that no specific query set was used for the TPC-H dataset. In fact, this dataset was used only in performance and scalability experiments, in which several synthetic query sets were used, allowing us to experiment with queries with much more keywords than those in the query sets reported in Table 8.3.

8.5 Golden Standards

To verify the effectiveness of our CNRank algorithm and its impact on the evaluation of CNs, it was necessary to use a set of relevant CNs and relevant JNTs for each of the tested queries. In the case of the Coffman-Weaver query set, the set of relevant JNTs is provided for each query in Coffman and Weaver [2010b]. To obtain the set of relevant CNs we simply took the CN that generates each relevant JNTs.

In the case of the SPARK and INEX query sets, we obtained the golden sets using the following procedure. For each query, we first, generated all CNs using the CNGen algorithm [Hristidis and Papakonstantinou, 2002]. Then, we manually evaluated all CNs to determine the relevant ones, which is how we obtained the CN golden sets. Next, we ran an SQL query based on each CN and obtained the resulting JNTs. The JNT golden set of a query is, then, the set of all JNTs generated by the CNs in its CN golden set.

8.6 Number of Relevant CNs

Our work is based on the hypothesis that the number of relevant CNs per query is typically much lower than the number of all possible CNs. Using the golden sets we generated, we could corroborate this hypothesis. In Table 8.4 we present the number of relevant CNs found per query, considering all queries and each query set individually.

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

TABLE 8.4: Number of relevant CNs per query.

Note that the maximum number of relevant CNs for any query is only two. In fact, all queries that have two relevant CNs are from the SPARK query set. The vast majority of queries (86% overall) have one single relevant CN. In the case of the Coffman-Weaver and INEX datasets, all queries had a single relevant CN. In fact, many queries had one single JNT as an answer. In Table 8.5, for each query set used, we compare the number of CNs generated with the number of relevant CNs. In each case, we show the maximum (MAX) number and the average (AVG) number of CNs.

Notice that number of relevant CNs is indeed much lower than the number of all CNs. There are cases in which the number of relevant CNs is less than 0.1% of all CNs.

Datasets	Generated CNs / Relevant CNs					
	Coffman-Weaver		SPARK		INEX	
	MAX	AVG	MAX	AVG	MAX	AVG
IMDb	62/1	18.0/1	42/2	25.9/1.5	163/1	33.4/1
Mondial	62/1	16.6/1	820/2	107.7/1.2		
Wikipedia	102/1	23.0/1				
DBLP			1531/2	255.8/1.6		

TABLE 8.5: Generated CNs vs. Relevant CNs.

Chapter 9

Experiments with CN Generation

In this chapter, we report a comprehensive set of experiments performed using several databases and query sets previously used in similar experiments reported in the literature. Our goal is to analyze two aspects related to the process of CN generation we propose: first, the quality of the set of CNs it produces, and, second, its performance and scalability. In addition, we also present results on how the CNs generated by MatCNGen impact the results obtained by well-know CN evaluation algorithms.

9.1 General Results

In this section we present numbers related to the volume of data handled while generating CNs. As scalability has been mentioned as an important issue in processing keyword queries [Baid et al., 2010], our goal here is to provide a perspective on how scalable is our method. These numbers are useful for validating the assumptions we made when designing our method and to explain the performance results we achieved.

Table 9.1 presents the maximum and the average number of query matches generated for each pair of query sets/datasets. The number of query matches is potentially larger for databases with many relations, as is the case in the Mondial dataset, and with many tuples as is the case in the IMDb dataset (see Table 8.1). Still, the average number of matches for the 218 queries we used is lower than 17.

As it can be observed, the number of query matches generated is small in most cases, with a few exceptions. For instance, the query *"South East"* from the SPARK query set over the Mondial dataset, yielded 208 query matches. Although this dataset is much smaller than the others, it has the highest number of relations, 28, and it has an intricate Schema Graph, since its schema

Dataset	Query Matches					
	Coffman-Weaver		SPARK		INEX	
	Max	Avg	Max	Avg	Max	Avg
IMDb	69	9.10	45	17.57	123	22.28
Mondial	16	4.20	208	23.20	-	-
Wikipedia	36	4.94	-		-	-
DBLP	-	-	6	2.00	-	-

TABLE 9.1: Number of query matches generated.

includes more than 100 Referential Integrity Constraints (see Table 8.1). This raises the number of ways tuple-sets can be combined to generate query matches.

In Figure 9.1, we compare the average number of CNs generated by our algorithms, **SingleCN** and **SteinerCN** , and by the baseline CNGen. The implementation of CNGen [Hristidis and Papakonstantinou, 2002] we used is available as part of the Efficient system [Hristidis et al., 2003], which was kindly made available by its authors.

Notice that our algorithms generated, on average, 69% less CNs than CNGen in all configurations of Query Sets and Datasets. This difference is very large in the case of DBLP, where they generated less than 10% of the CNs generated by CNGen. The fact that our algorithms generate less CNs than CNGen for all query sets with all databases was expected, since CNs are generated for each query match. Thus, the number of CNs is proportional to the number of query matches for a given query. Obviously, this has a positive impact on the overall performance and scalability, as discussed in Section 9.3. Interestingly, as we will discuss in detail in Section 9.2, this smaller set of CNs is of high quality, yielding results at least as good as those obtained with the larger number of CNs generated by CNGen.

9.2 Quality Results

In this section we study how our method impacts the quality of the results of processing keyword queries. Our evaluation consists of measuring the quality of the output produced by the Hybrid [Hristidis et al., 2003] and the Skyline Sweeping [Luo et al., 2007a] algorithms, when taking as input the set of CNs generated by our algorithms. Both are a well-known algorithms used for evaluating a set of CNs over a database, providing as results a set of joining networks of tuples (JNT). The implementations we use here were kindly provided by their respective authors. We report results obtained with four configurations: MatCNGen+SingleCN with Hybrid (MCG+H), MatCNGen+SingleCN with Skyline Sweeping (MCG+SS),MatCNGen+SteinerCN with Hybrid (SCG+H) and MatCNGen+SteinerCN with Skyline Sweeping (SCG+SS).

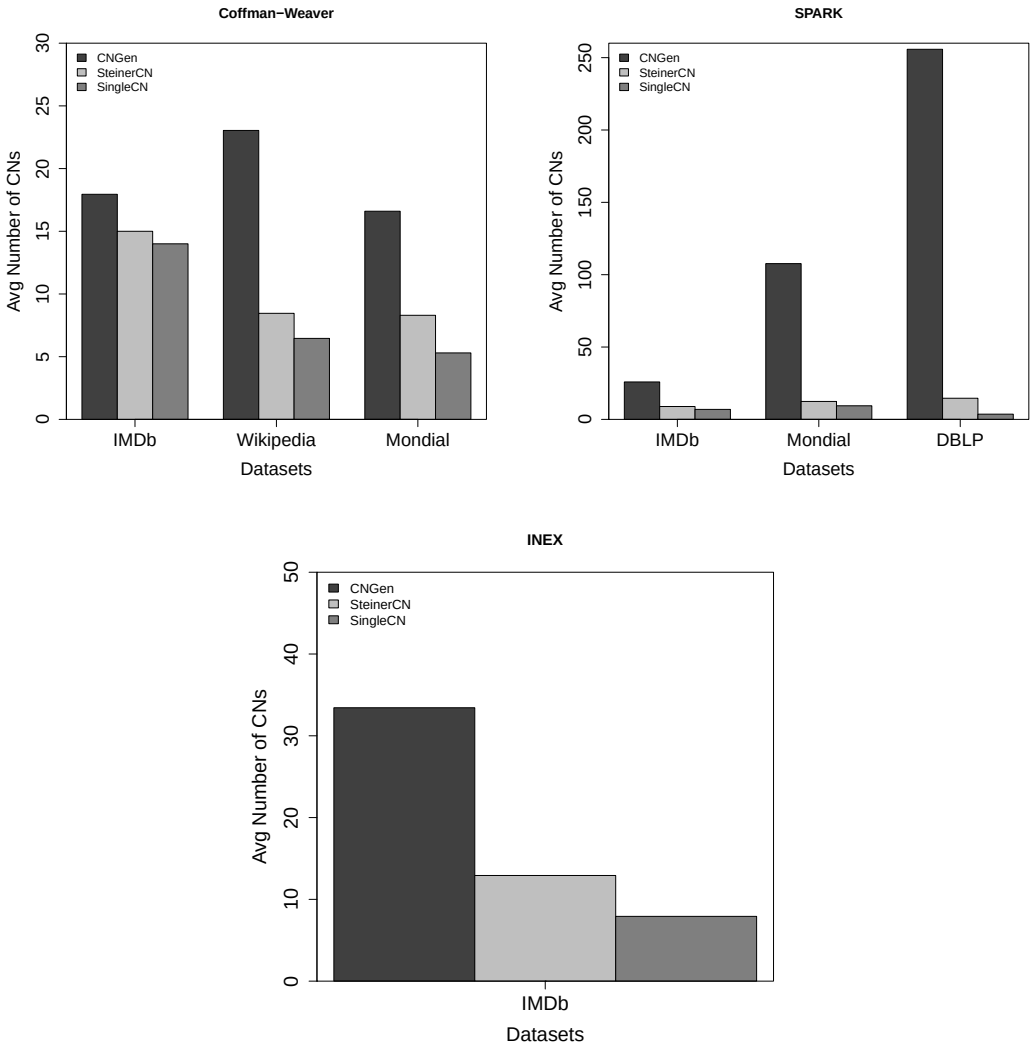


FIGURE 9.1: Average number of CNs generated for all query sets and datasets.

9.2.1 Quality Metrics

To evaluate the results produced by each system/configuration for each query set, we used the well-know *Mean Average Precision (MAP)* metric [Baeza-Yates and Ribeiro-Neto, 1999]. Let A be a ranking of the answers generated for a given keyword query Q. The *Average Precision* (AP_Q) of this ranking is the average of the precision values calculated at each position k in which there is a relevant JNT in A. That is, $AP_Q = \sum_{k=1}^{n} P(k) \times rel(k)/|R|$, where n is the number of JNTs considered from the ranking A (in our case, $n = 1000$), $P(k)$ is the precision at position k, $rel(k)$ is 1 if the answer at position k is relevant or 0 otherwise, and R is the set of know relevant JNTs for Q. Then, MAP value is the average of AP_Q, for all queries Q in a given query set.

On the other hand, in all query sets there are some queries that have only one relevant JNT as answer. For evaluating how each system handles these queries specifically, we used the *MRR*

(Mean Reciprocal Rank) metric [Baeza-Yates and Ribeiro-Neto, 1999], which is more adequate for these cases. Given a keyword query Q, the value of RR_Q, called *Reciprocal Ranking*, is given by $\frac{1}{K}$, where K is the rank position of the first relevant JNT. Then, the MRR value is obtained for all queries in a query set is the average of RR_Q, for all Q in the query set. In our case, the MRR value indicates how close the correct JNT is from the first position of the ranking generated by each system.

9.2.2 Results – Coffman-Weaver Query Set

In Coffman and Weaver [2010a] report results of quality experiments carried out with queries from the query set they generated using several R-KwS systems previously presented in the literature: BANKS [Aditya et al., 2002], DISCOVER [Hristidis and Papakonstantinou, 2002], Efficient [Hristidis et al., 2003], Bi-directional [Kacholia et al., 2005], Effective [Liu et al., 2006], DPBF [Ding et al., 2007], BLINKS [He et al., 2007], SPARK [Luo et al., 2007a] and CD [Coffman and Weaver, 2010c]. These two authors gently provided us their results, along with the set of relevant tuples that should be returned by each query, which were used as the gold standard to measure the quality of the results obtained by each system. These resources allowed us to compare our four configurations: MCG+H, MCG+SS, SCG+H and SCG+SS with these systems.

The overall results for all queries of the Coffman-Weaver query set are presented in Figure 9.2 in terms of MAP. For this query set, 111 out of 129 keyword queries, have only one relevant JNT as answer. The MRR values achieved by each system when processing these queries are presented in Figure 9.3.

9.2.3 Results – Spark and INEX Query Sets

In the case of queries from the Spark and INEX query sets, we did not had access to the same resources as for the Coffman-Weaver query set. Thus, we first generated the golden standards for all queries of both datasets by ourselves, based on the query descriptions available. Then, we run quality experiments using configurations obtained by coupling CNGen with Hybrid and with Skyline Sweeping as baselines. Simply for the sake of symmetry, we named these configurations CNGen+H and CNGen+SS, respectively. Notice, however, that they in fact correspond to systems DISCOVER [Hristidis and Papakonstantinou, 2002] and Efficient [Hristidis et al., 2003], respectively.

The results are presented in Figure 9.4. The graph on the left presents the results in terms of MAP achieved for all queries in each query set. The graph on the right presents the MRR results

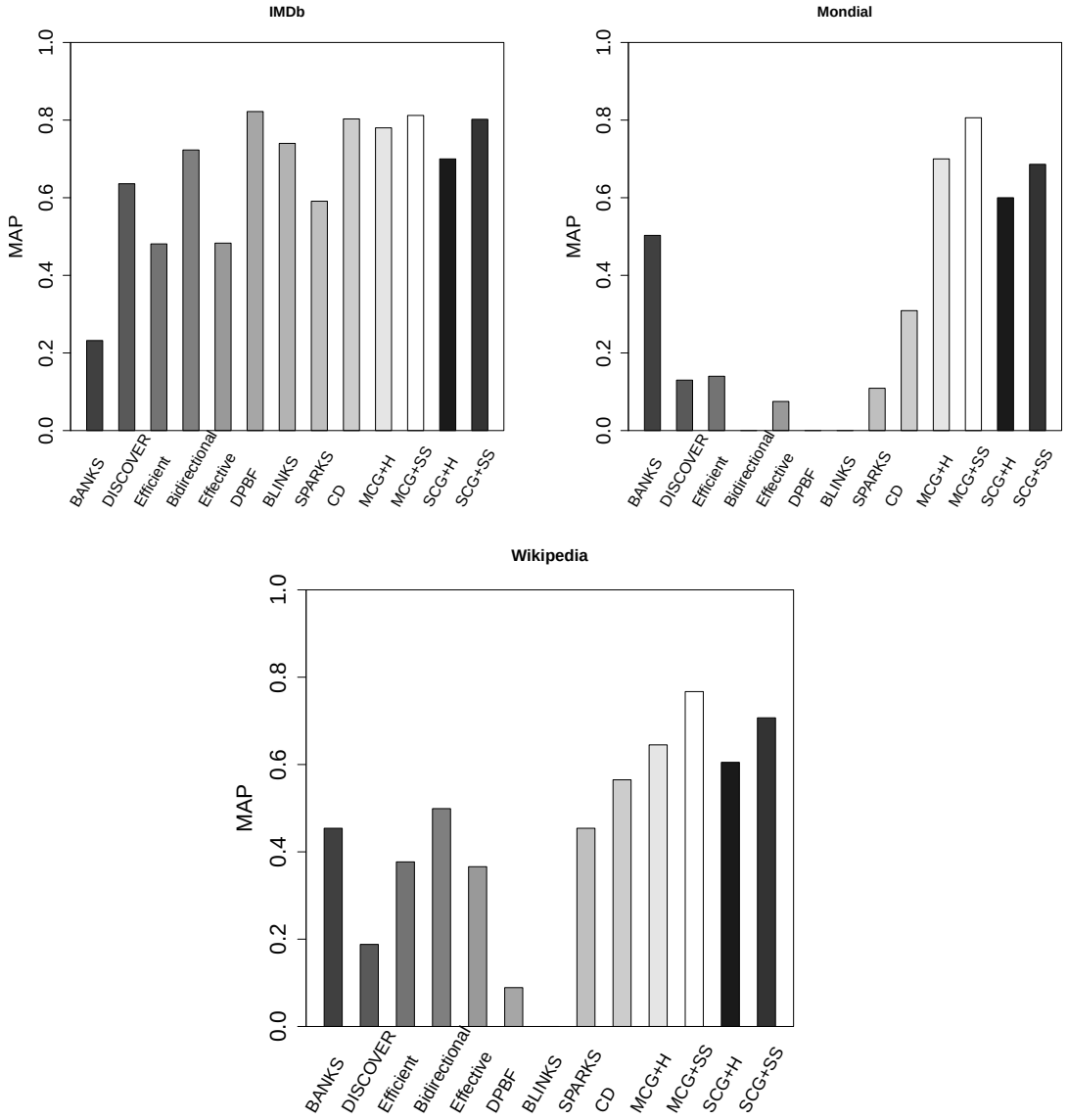


FIGURE 9.2: MAP measured across the various systems and datasets for queries from Coffman-Weaver.

achieved only for queries that return a single JNT as result. There were 63 of such queries for the SPARK query set and 11 for the INEX query set.

9.2.4 Analysis

In the experiments with the Coffman-Weaver query set, the configurations based on MatCNGen achieved the best results compared with the other methods in all datasets, with slight advantage for the configuration that uses SkylineSweeping. For Mondial and Wikipedia, the MAP values obtained by these configurations were much higher than the other systems. Only for the IMDB dataset we observed a small gain in comparison to the third best result, obtained by DPBF.

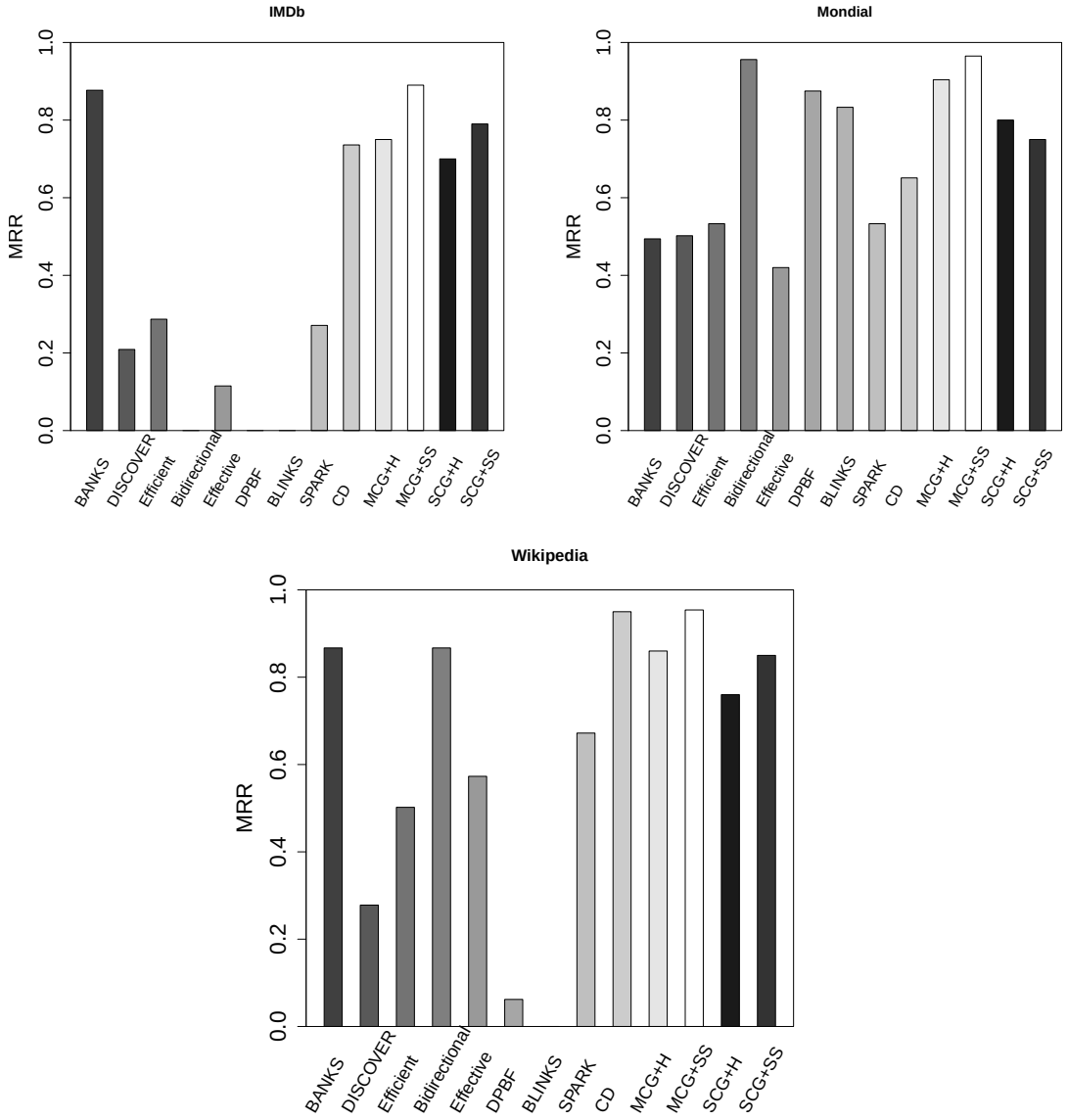


FIGURE 9.3: MRR results for each system for queries from Coffman-Weaver where exactly one JNT is relevant.

In the case of the experiments with the SPARK and INEX query sets, again the configurations based on MatCNGen presented better results, with slight advantage for MCG+SS, in comparison with those achieved by configurations based on CNGen. In a single case, IMDb/SPARK, MCG+SS outperformed MCG+H,SCG+H and SCG+SS in terms of MAP.

An important observation regarding the results obtained with all three query sets, across all datasets, is that MatCNGen, in spite of generating less CNs than CNGen, led to the best quality of results in all tests we performed. This corroborates our claims regarding the quality of the CNs generated and their potential of positively impact in results produced by RwK-S systems.

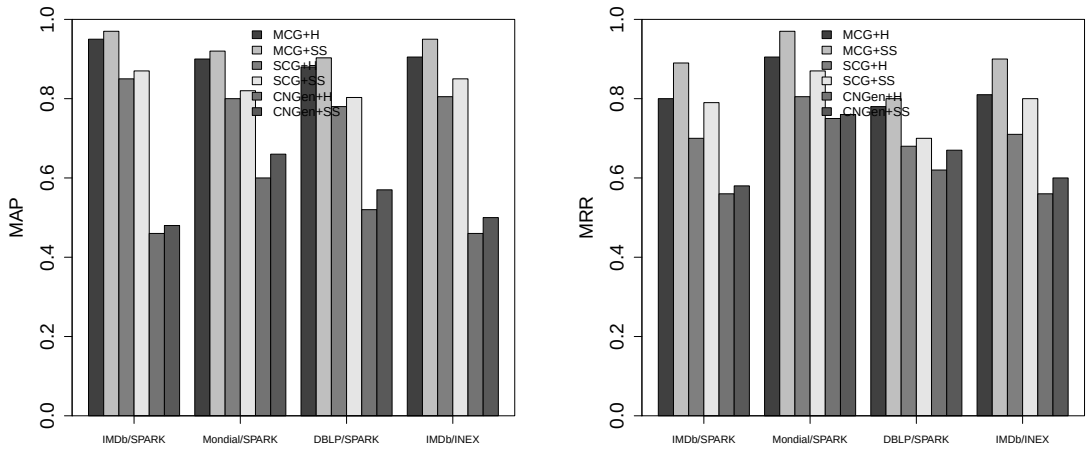


FIGURE 9.4: MRR and MAP measured across the various systems and datasets for queries from SPARK and INEX.

9.3 Performance and Scalability Results

In the experiments presented in this section, our goal was to verify the performance and the scalability of our method for generating Candidate Networks as it is used in several other works [Coffman and Weaver, 2010b]. We use as a baseline the implementation CNGen [Hristidis and Papakonstantinou, 2002] available on the Efficient system [Hristidis et al., 2003]. To the best of our knowledge, CNGen is still the state-of-the-art for Candidate Network generation, as it is used in a number of papers [Coffman and Weaver, 2010c; Hristidis et al., 2003; Hristidis and Papakonstantinou, 2002; Liu et al., 2006; Luo et al., 2007a]. As this implementation was originally target to the ORACLE DBMS, we had to adapt it to work with PostgresSQL, which we used in our implementations. Thus, all implementations tested run under exactly the same operational conditions.

For these experiments, we used two different versions of MatCNGen: *MatCNGen-Disk*, called MCG-D, in which the generation of tuples is carried out using the disk-based version of the TS-Find algorithm (Figure 6.2); and *MatCNGen-Mem*, called MCG-M, which uses the counterpart memory-based version (Figure 6.5).

9.3.1 Overall Results

Figure 9.5 compares, for all queries in each query set, the average time spent for generating the corresponding Candidate Networks, using each implementation tested. Each bar in this graph represents the average time spent by a given system for generating CNs for the queries of a query set targeted to a dataset, as identified in the top of the graph. In each bar, we separate the

time it takes to generate the tuple-sets (e.g., CNGen/TS) from the time spent with the process of constructing the CNs itself (e.g., CNGen/CN). This allowed us to separately assess the impact in processing time due to each version of the TSFind algorithms, from the impact due to the MatchCN algorithm, used for the task of obtaining CNs from match graphs. Notice that in the case of MatchCN, the time also includes the generation of query matches. We report only the results obtained with the SingleCN algorithm, but the results using SteinerCN follows a similar trend.

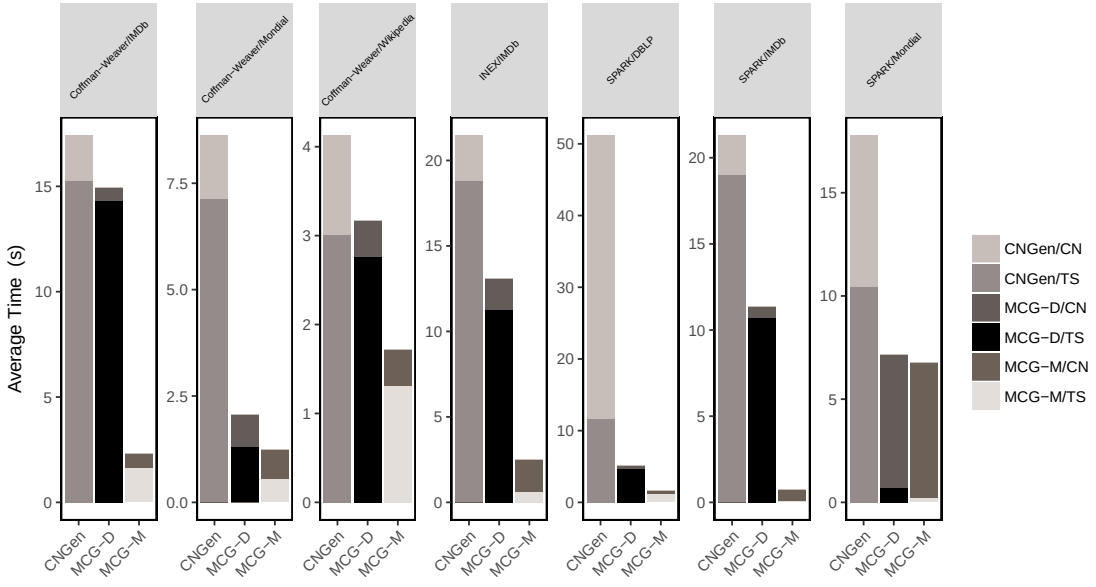


FIGURE 9.5: Average time to generate CNs using CNGen, MatCNGen-Disk (MCG-D) and MatCNGen-Mem (MCG-M).

As it can be noticed, both MatCNGen implementations outperformed CNGen in all cases. Specifically for the task of generating tuple-sets, MatCNGen-Disk is faster than CNGen, but, of course, MatCNMen-Mem is by far faster than both. As expected, this advantage is greater when dealing with datasets that have many relations, i.e., IMDb, Wikipedia and DBPL, since potentially more tuple sets must be generated for each query.

Regarding the time to assemble CNs, as expected, the fact that the MatchCN algorithm builds a singe CN for each query match resulted in a time improvement in all cases. This impact is higher in the cases such as SPARK/DBLP, which has a few query matches (see Table 9.1), than in cases in which many query matches exist, as in SPARK/Mondial.

9.3.2 Scalability with the Number of Keywords

A well-known drawback in current R-KwS systems is their scalability with the number of keywords in queries. This issue has been studied in the literature [Baid et al., 2010; Markowetz

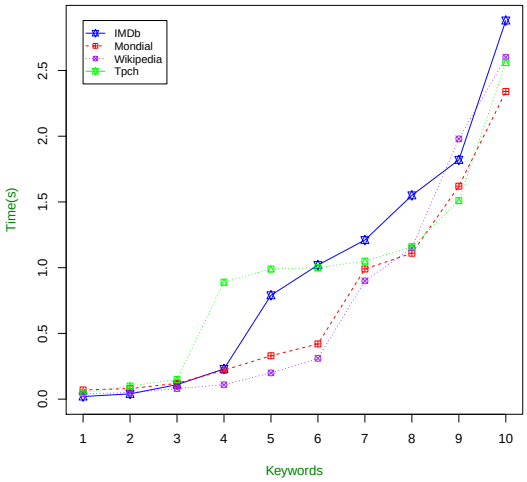
et al., 2007], since queries with a high number keywords usually cause an excessive memory consumption.

In the experiments we reported so far, we have not faced this issue, since, as shown in Table 9.1, the maximum number of keywords found in the query sets we used is four. Thus, to study the behavior of our method when the number of keywords grows beyond this value, we had to generate new query sets, as follows. For each dataset we randomly generated a load of 100 queries with K keywords, varying K from one to ten. Each query was then submitted to CNGen and MatCNGen and we measure time spent by each method to generate CNs. In the case of MatCNGen, we have used only the MatCNGen-Mem, since this implementation demands much more memory than MatCNGen-Disk. The results are shown in Figure 9.6. In Figure 9.6 (a) and (b), each curve corresponds to queries issued over datasets IMDB, Mondial, Wikipedia and TPCH, and each point corresponds to the average time spent with 100 queries for each value of K.

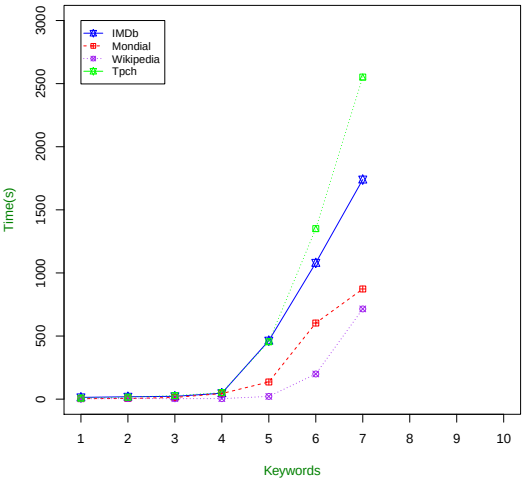
The results for the DBLP dataset are shown separately in Figure 9.6 (c), where the Y axe (time) is in log scale to accommodate the disparate scales of time values obtained with CNGen in comparison to MatCNGen. In the case of CNGen, we observe a poor scalability. As show in Figures 9.6 (b) and (c), the system could not process any query with more than seven keywords with the computational setup used in the experiments, since the implementation we used crashed after this. This same behavior was observed by the authors in Baid et al. [2010]. In fact, we observed the same failures with many other queries from four keywords on. For instance, with five keywords, about half of the queries caused the system to crash. In such case, we simply removed this query from the time average. It is worth noticing that we observed no failures with MatCNGen, whose actual times for all the queries are presented in Figure 9.6 (a) and (c). It must be highlighted that although this experiment indicates that MatCNGen is able to handle queries with a larger number of keywords, there is a consensus in the literature that queries with more than four keywords are very unlikely and that very often queries have two or just one keyword.

9.3.3 Discussion

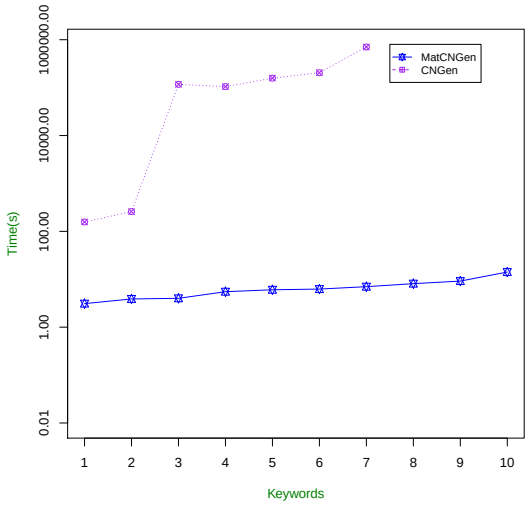
Previous work in the literature have reported that processing Keyword Queries over relational databases has unpredictable running times, with certain queries taking too long to run or even failing due to memory exhaustion [Baid et al., 2010] [Coffman and Weaver, 2010a]. Indeed, the generation of Candidate Networks can be a costly operation regardless of the method used. For instance, on average, the SPARK query set issued over the Mondial dataset took 1.51 seconds to generate CNs using MatCNGen-Mem. Nevertheless, the memory-based version of our method makes it viable to generate keyword queries on-line. For instance, CNGen took more than fifty seconds, on the average, to generate the CNs for the queries over the DBLP dataset, which



(a) MatCNGen-Mem



(b) CNGen



(c) DBLP

FIGURE 9.6: Average time spent to generate Candidate Networks when varying the number of keywords.

can be too much for on-line applications. On the other hand, our memory version took about 0.5 seconds in this case. It is important to note that the significant performance gains we have achieved do not compromise the quality of the results obtained, as discussed in Section 9.2.

Chapter 10

Experiments with CN Ranking

In this chapter we present experimental results on the effectiveness of our algorithms in the task of ranking Candidate Networks given a keyword query. All the experiments in this chapter use sets of Candidate Networks generated by the CNGen algorithm [Hristidis and Papakonstantinou, 2002] instead of the ones generated by our own algorithms. We decided to do so for two main reasons. First of all, as we had already shown, CNGen generates much more CNs then our algorithms. Thus, the task of ranking these CNs is considerably more difficult. Second, we wanted to remark that CNRank is independent of our CN generation process, as it is from any other CN generation method.

10.1 General Results

In Figure 10.1, we present an evaluation of the ranking produced by CNRank using the MRR (Mean Reciprocal Rank) metric. Given a keyword query Q, the value of RR_U is given by $\frac{1}{Q}$, where Q is the rank position of the first relevant CN in the ranking. Then, the MRR obtained for queries in a query set is the average of RR_Q, for all Q in the query set. Intuitively, the MRR metric measures how close relevant CNs are from the first position in the ranking generated by CNRank.

In summary, the results in Figure 10.1 show that CNRank is able to place the relevant CNs in the top positions of the rank, provided that CNGen generates this target CN. For five of the seven combinations of query set/datasets, MRR values were above 0.9. In the two remaining cases, MRR was also high: 0.81 for SPARK/IMDb and 0.82 for Coffman-Weaver/Wikipedia.

Next, we present in Figure 10.2 the precision levels achieved by CNRank using the $P@K$ (precision at position K) metric. Given an keyword query U, the value of $P_U@K$ is one if the target query for U appears in a position up to K in the ranking, and zero otherwise. $P@K$ is the

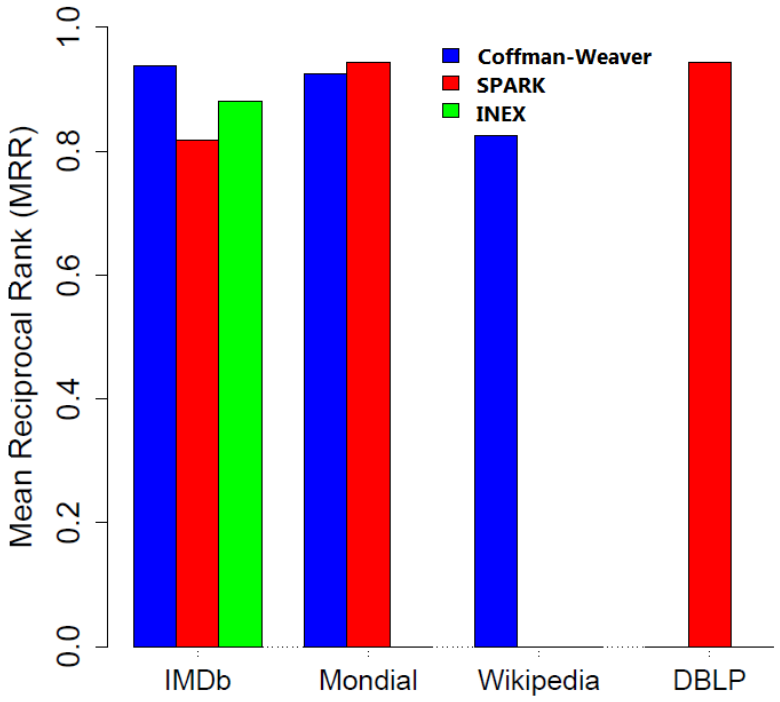


FIGURE 10.1: MRR values achieved by CNRank.

average of $P_U@K$, for all U in a query set. Figure 10.2 presents $P@1$ to $P@4$ results for all query sets.

First of all, we observed that in all query sets, the correct answers was always among the top-4 positions in the ranking, since P@4 is one in all cases. Considering all queries, on average, $P@1$ is 0.83, which means that the CNRank very often assigns a relevant CN to the top-1 position. Importantly, in all cases, $P@1$ was no lower than 0.63. Also, notice that P@2 is one in all queries from the SPARK query set.

We noticed that in many cases the inherent ambiguity of certain queries was harmful to the evaluation we adopted. For instance, for the query *"fast food"*, the information need statement provided by INEX requires a movie in the answer. For this query, the CNRank algorithm placed the CN that provides the required answer in the second position. However, we found out that the CN assigned by CNRank to the first position, which retrieves a character instead, is also plausible and provides a suitable answer.

10.2 Impact on CN Evaluation

As we have already shown, by using CNRank, we can drastically reduce the number of CNs that are evaluated. That is, instead of evaluating tens or hundreds of CNs as in current R-KwS

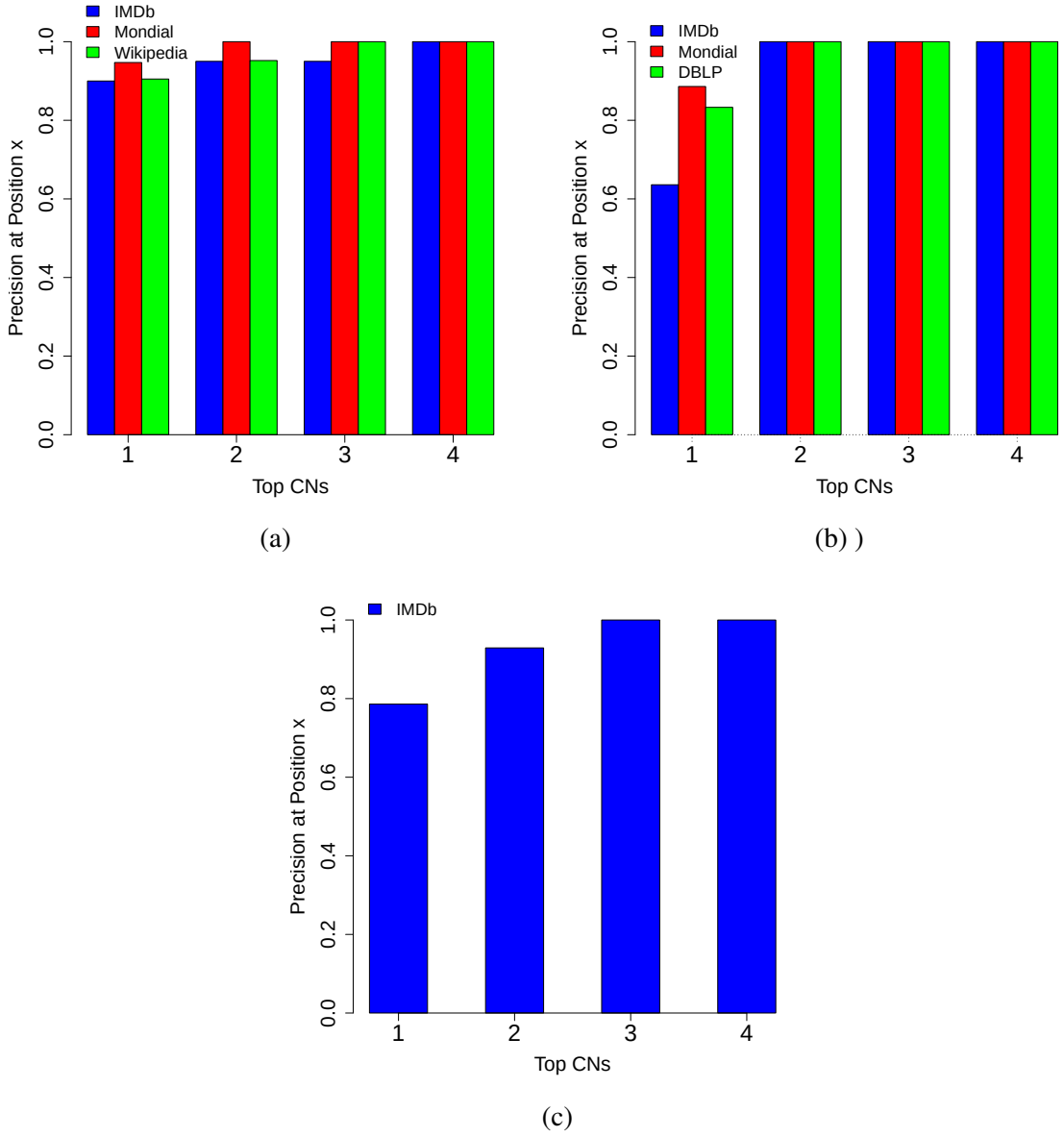


FIGURE 10.2: P@k values achieved by CNRanking on Coffman-Weaver (a) SPARK (b) and INEX (c).

systems, only a few top CNs selected using CNRank need to be evaluated. In this section we show experiments we performed to verify the impact of using only a few top CNs in evaluating CNs to produce the JNTs that comprise the answer for an unstructured query.

For this experiment, two distinct evaluation algorithms were used, namely, Hybrid [Hristidis et al., 2003] and Skyline-Sweeping [Luo et al., 2007a]. These evaluation algorithms were the best among those proposed in their respective papers and are highly representative of the state of the art in CN evaluation. We compared the results obtained by each algorithm in three different configurations considering the input they receive: (1) all generated CNs, identified respectively as “Hybrid” and “Skyline”; (2) only the top-4 CNs selected with CNRank, identified respectively

as “HybRank4” and “SSRank4”; and (3) only the top-1 CN selected with CNRank, identified respectively as “HybRank1” and “SSRank1”. The decision of using the top-4 CNs was suggested by the results from Figure 10.2, where we have shown that the relevant CNs were among the top-4 CNs ranked by CNRank. In Figure 10.3, we present a comparison of the results obtained with each of these configurations using the MRR metric.The MRR values were calculated in a way similar to that described above for evaluating CNs, but this time we applied it to evaluating the ranking of JNTs.

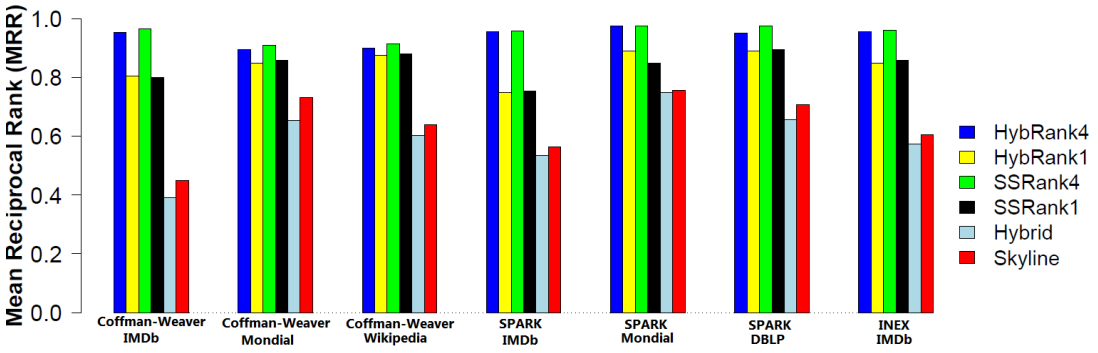


FIGURE 10.3: Effect of CNRank on CN evaluation in terms of MRR.

The graph in Figure 10.3 shows that using CNRank yields high MRR values in all query sets. Indeed, these values are much higher than the values obtained without CNRank. This corroborates our claims regarding the positive impact of using CNRank in CN evaluation. This was observed for both Hybrid and Skyline-Sweeping, which suggests that CNRank can improve the results of any CN evaluation algorithm. Indeed, this occurs due to the fact that now the CN evaluation algorithm receive only those CNs that are likely to produce relevant JNTs.

Notice that the configurations that use top-4 CNs achieved better results than the top-1 configurations. This is expected, since top-1 configurations may miss relevant CNs. Nevertheless, the top-1 configurations performed better than the baselines. Incidentally, notice that the Skyline-Sweeping performed better than the Hybrid algorithm. However, their counterparts that use CNRank provided very similar results.

Coffman and Weaver [2010a] performed similar experiments also using the MRR metric. In this case, however, the authors only used queries that returned a single JNT as results. The goal was to evaluate how well this single-answer JNT is placed in the ranking. We also ran this same experiment with the configurations we implemented. To distinguish this metric, which applies to a single answer, from the one used in Figure 10.3, which applies to multiple answers as well, we called this metric *SMRR*. The result of this experiment is presented in Figure 10.4. Performing this experiment allowed us to compare our results with those obtained by a number of other R-KwS systems evaluated by Coffman and Weaver [2010a],and whose detailed numbers were generously provided by its authors. This includes not only systems based on the Schema Graph

approach, i.e., DISCOVER [Hristidis and Papakonstantinou, 2002], Efficient [Hristidis et al., 2003], Bidirectional [Kacholia et al., 2005], Effective [Liu et al., 2006], and CD [Coffman and Weaver, 2010c], but also in systems based on the Data Graph approach, i.e., BANKS [Aditya et al., 2002], DPBF [Ding et al., 2007] and BLINKS [He et al., 2007]. In this case, all results are from the Coffman-Weaver query set, which was the only one used in the experiments reported by Coffman and Weaver [2010a].

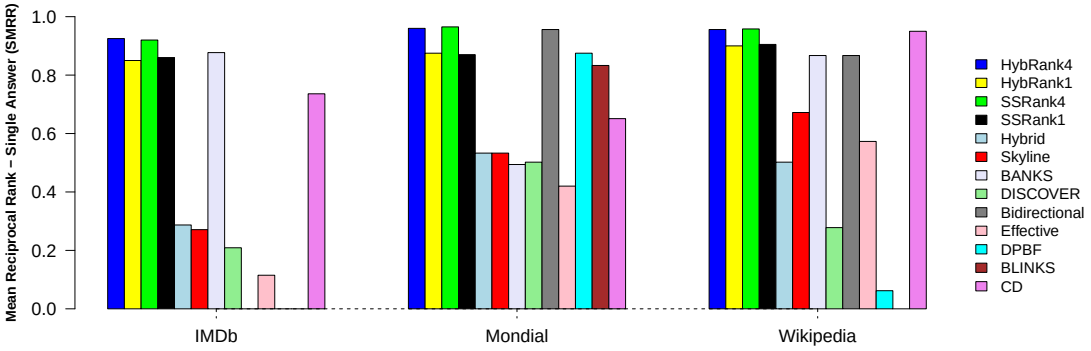


FIGURE 10.4: Impact of CNRank. SMRR on Coffman-Weaver query set.

Again, the configurations based on CNRank provided superior values of SMRR, with a slight advantage to configurations that receive top-4 CNs. These configurations had results equal to the best system in each dataset, that is, BANKS in IMDb, Bidirectional in Mondial and CD in Wikipedia. It should be noted that none of the tested system performed consistently well across all datasets. However, such a trend was observed in all CNRank-based configurations.

To evaluate the impact of CNRank in the overall quality of the ranking, we used the MAP (Mean Average Precision). Let A be a ranking of JNTs generated for a given keyword query U. The *Average Precision* (AP_U) of this ranking is the average of the precision values calculated at each position k in which there is a relevant JNT in A. That is, $AP_U = \sum_{k=1}^{n} P(k) \times rel(k)/|R|$, where n is the number of JNT considered from the ranking A (in our case, $n = 1000$), $P(k)$ is the precision at position k, $rel(k)$ is one if the answer at position k is relevant or zero otherwise, and R is the set of known relevant answers for U. Then, the *Mean Average Precision (MAP)* is the average of AP_U, for all U in a query set.

In Figure 10.5, we present the MAP values obtained with the configurations we implemented. Again, for the case of the Coffman-Weaver query set, it was possible to compare our results with those reported by Coffman and Weaver [2010a], as shown in Figure 10.5 (right).

The MAP results in Figure 10.5 reveal that CNRank also affects the global ranking quality in all tested scenarios. Again, the CNRank-based configurations consistently achieved better

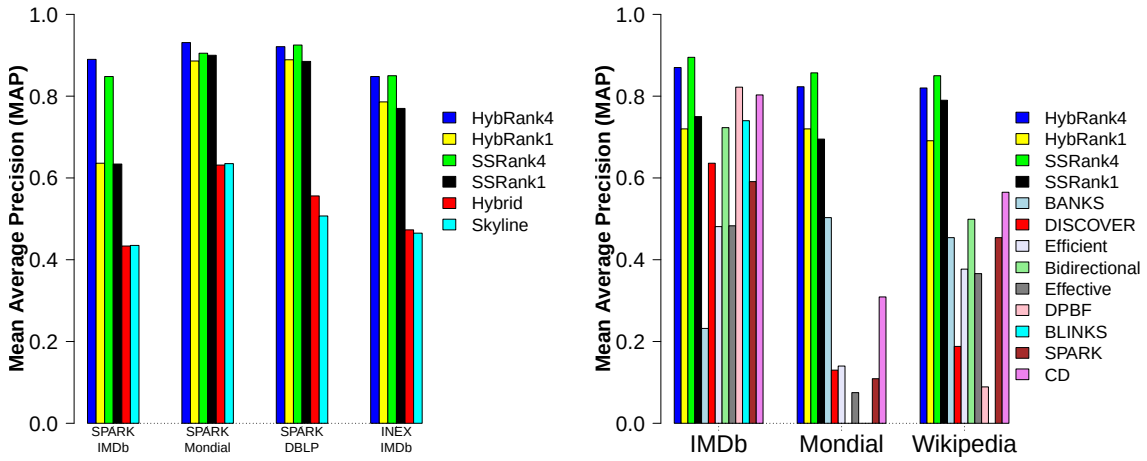


FIGURE 10.5: Impact of CNRank on CN evaluation in terms of MAP – SPARK and INEX (left), Coffman-Weaver (right).

results across different query sets and datasets, compared to all previously proposed methods considered.

10.3 Impact on Performance

An important question regarding introducing CNRank in the processing of keyword queries in R-KwS systems is how it affects their performance. In Figure 10.6 we present, for each query set/dataset pair, the average time spent processing a keyword query. In the case of ”Hybrid” and ”Skyline” configurations, this time includes the generation and evaluation of CNs. In the case of ”HybRank4”, ”SSRank4”, ”HybRank1” and ”SSRank1” this time also includes the ranking of CNs performed by CNRank. The graph in Figure 10.6(left) includes all query set/dataset pairs, except for SPARK/DBLP, whose values required a different scale. We decided to show this result in a separate graph in Figure 10.6(right).

Figure 10.6 shows that, as expected, CNRank also positively affects performance. Obviously, it is due to the fact that CN evaluation algorithms now have much less CNs to handle. Also, these graphs show that the process of ranking CNs introduced between the generation and evaluation of CNs does not imply any significant overhead to the whole process.

We wish to point out that the Skyline-Sweeping algorithm performed better than Hybrid, which is consistent with the results presented by Luo et al. [2007a]. This trend was also observed, on a smaller scale, in the CNRank-based configurations that use each algorithm.

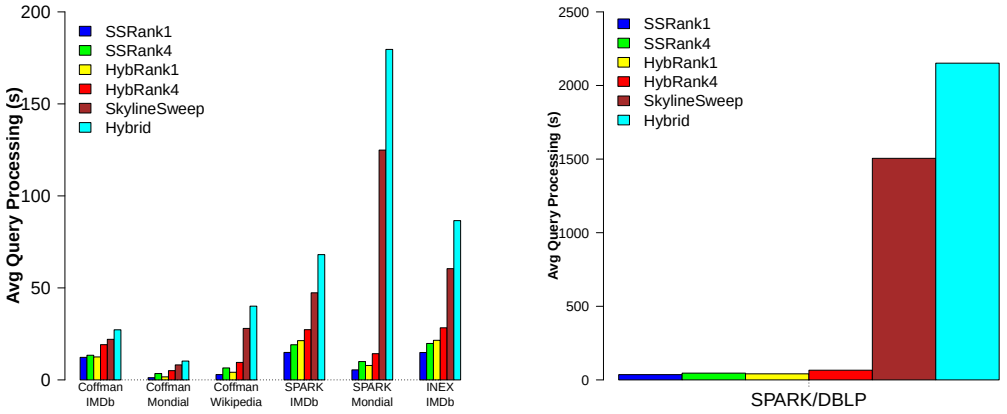


FIGURE 10.6: Impact of CNRank on CN evaluation – Performance.

Interestingly, we did not observe a significant advantage, in terms of performance, of the configurations that use only the top-1 CN over those that use top-4 CNs. This suggests that top-4 CN configurations, which in the previous experiments resulted in a better ranking in terms of quality, are preferable.

Chapter 11

Conclusions and Future Work

11.1 Conclusions

In this thesis we presented two contributions for the general problem of processing keyword queries over relational databases by means of Candidate Networks (CNs). First, we presented a novel approach we have developed for generating CNs. Our approach, called *Matching Graph Induced-based of Candidate Network Generation*, or *MatCNGen*, enabled the development of efficient and effective algorithms that are able to generate a compact, but optimized set of CNs that lead to superior quality answers, while demanding less resources in terms of processing time and memory used.

As our second main contribution we proposed an approach for ranking CNs. Our motivation for this approach comes from evidences we collected, and also reported here, that, although the number of possible CNs can be very high, only very few of them in fact produce answers relevant to the user and are indeed worth processing. In comparison to traditional R-KwS systems that process all possible CNs, processing only top-ranked CNs yields to improve not only the time it takes to return answers and but also the quality of the answers retrieved.

Our claims are supported by a comprehensive set of experiments we carried out using several query sets and datasets previously used by other works. In particular, we have shown that our approach makes it viable for R-KwS system processing of keyword queries in a on-line setting. So far, previous work in the literature have reported this task had unpredictable running times, with certain queries taking too long to run or even failing due to memory exhaustion.

11.2 Future Work

Although the results we reached corroborate our initial claims and are very promising towards the implementation of scalable R-KwS systems, there are several research directions yet to be explored and that are probably out of the scope of this thesis. We list below some of the topics we could identify so far, in no particular order.

- **Supporting structural references in keyword queries.** Currently, we only consider keywords which refer to database instances (i.e., contents). It would be interesting to consider cases in which the user may have some knowledge of the schema, while she is specifying the keywords. Specifically, the names of some attributes may be known or are easy to guess. In this case, queries might include such structural hints as done by Cohen et al. [2003] and this could be used to further improve our results, both in CNs generation and CN ranking.

- **Addressing CN evaluation.** The bayesian framework we used to rank CNs can be also used to rank the tuple networks that result from evaluating the CNs against a database instance. This has been already experimented for a similar task in LABRADOR [Mesquita et al., 2007]. Thus, a possible future work is to adapt our bayesian network to deal with generic tuples of network, and to compare this approach to previous methods in the literature for CN evaluation.

- **Considering alternative query semantics.** Currently, both methods we presented assume that queries use an "AND" semantics. However, our approach for generating CNs make it plausible to also handle alternative semantics. A natural candidate would be, of course, the "OR" semantics, but a more interesting semantics for our context might be the so-called "zero-recall" semantics [Singh et al., 2011]. In this semantics, a query is initially assumed to have an "AND" semantics, but if it returns no result, the query is rewritten by removing some of its terms, so that some meaningfull can be obtained.

- **Considering alternative weighting schemes.** Currently, our weighting scheme used for calculating CN scores for ranking is based only on features related to the lexical properties of the atributes, i.e., on measuring which terms are more representative for a given attribute. Although we have achieved good results with this scheme, it would be interesting to investigate other weighting schemes based on other features. We list some of them below:

  - The relative and absolute position of keywords in the query. This feature has been extensively used in the context of information extraction [Cortez et al., 2010, 2011] and query structuring [Li et al., 2009; Sarkas et al., 2010], and could be also considered in the context of our target problems.

  - Statistics on key/foreign key instances. These statistics may indicate that certain arcs in the schema graph are more likely to be observed in actual data than others, and this information can be used for enhancing the ranking of CNs.
  - User preferences towards certain attributes. In certain settings, users may prefer some attributes over the others. These preferences may explicitly declared or inferred from query logs and can also be used in the ranking model.

- **Using alternative approaches to the Bayesian Framework.** Another possible future line of investigation is exploring alternatives to the Bayesian framework for combining the individual probabilities estimated for the attributes. This line is particularly interesting if the number of features to consider increases, as suggested above. In this case, we believe that approaches based on machine learning are worth being considered.

- **Alternative approximated algorithms for Steiner trees.** We have used a specific approximation algorithm to compute Minimal Steiner Tress. However, there are other alternatives in the literature that could be explored and compared with the one we currently use.

- **Enabling keyword queries in a DBMS.** Our algorithms can be used to allowing the processing of keyword queries directly into a DBMS. A number of benefits may arise from such an implementation. Among them we cite: inherent availability of DBMS resources (e.g., ACID properties, optimization for query processing, etc.), potential improvement in performance due to the tighter coupling between our methods and the DBMS; natural portability of our methods to the same platforms as the DBMS; use of statistics already collect by DBMS components and the possibility of using triggers to maintaining the indexes used by our methods.

Bibliography

Aditya, B., Bhalotia, G., Chakrabarti, S., Hulgeri, A., Nakhe, C., Parag, P., and Sudarshan, S. (2002). Banks: Browsing and keyword searching in relational databases. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 1083–1086. VLDB Endowment.

Agrawal, S., Chaudhuri, S., and Das, G. (2002). DBXplorer: A system for keyword-based search over relational databases. In *Proceedings of the 18th International Conference on Data Engineering*, pages 5–, Washington, DC, USA. IEEE Computer Society.

Baeza-Yates, R. A. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Baeza-Yates, R. A. and Ribeiro-Neto, B. A. (2011). *Modern Information Retrieval - the concepts and technology behind search*. Pearson Education Ltd., Harlow, England, 2nd edition.

Baid, A., Rae, I., Li, J., Doan, A., and Naughton, J. (2010). Toward scalable keyword search over relational data. *Proceedings of the VLDB Endowment*, 3(1-2):140–149.

Bao, Z., Zeng, Y., Jagadish, H., and Ling, T. W. (2015). Exploratory keyword search with interactive input. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 871–876, New York, NY, USA. ACM.

Barros, E. G., Laender, A. H. F., Moro, M. M., and da Silva, A. S. (2016). Lca-based algorithms for efficiently processing multiple keyword queries over XML streams. *Data and Knowledge Engineering*, 103:1–18.

Barros, E. G., Moro, M. M., and Laender, A. H. F. (2010). An Evaluation Study of Search Algorithms for XML Streams. *Journal of Information and Data Management*, 1(3):487–502.

Coffman, J. and Weaver, A. C. (2010a). A framework for evaluating database keyword search strategies. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, pages 729–738, New York, NY, USA. ACM.

Coffman, J. and Weaver, A. C. (2010b). Relational keyword search benchmark. http://www.cs.virginia.edu/ jmc7tp/resources.php.

Coffman, J. and Weaver, A. C. (2010c). Structured data retrieval using cover density ranking. In *Proceedings of the 2Nd International Workshop on Keyword Search on Structured Data*, pages 1:1–1:6, New York, NY, USA. ACM.

Cohen, S., Mamou, J., Kanza, Y., and Sagiv, Y. (2003). Xsearch: A semantic search engine for xml. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, pages 45–56. VLDB Endowment.

Cortez, E., da Silva, A. S., Gonçalves, M. A., and de Moura, E. S. (2010). Ondux: on-demand unsupervised learning for information extraction. In *SIGMOD Conference*, pages 807–818.

Cortez, E., Oliveira, D., da Silva, A. S., de Moura, E. S., and Laender, A. H. (2011). Joint unsupervised structure discovery and information extraction. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 541–552, New York, NY, USA. ACM.

Cyganiak, R. (2007). Benchmarking D2RQ v0.1.

da C. Hummel, F., da Silva, A. S., Moro, M. M., and Laender, A. H. F. (2011). Multiple keyword-based queries over xml streams. In *Conference on Information and Knowledge Management*, pages 1577–1582.

de Oliveira, P., da Silva, A., and de Moura, E. (2015). Ranking candidate networks of relations to improve keyword search over relational databases. In *2015 IEEE 31st International Conference on Data Engineering*, pages 399–410.

Diestel, R. (2012). *Graph Theory*. Springer, 4th edition.

Ding, B., Yu, J. X., Wang, S., Qin, L., Zhang, X., and Lin, X. (2007). Finding top-k min-cost connected trees in databases. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 836–845.

Dreyfus, S. E. and Wagner, R. A. (1971). The steiner problem in graphs. *Networks*, 1(3):195–207.

He, H., Wang, H., Yang, J., and Yu, P. S. (2007). Blinks: Ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 305–316, New York, NY, USA. ACM.

Hearne, T. and Wagner, C. (1973). Minimal covers of finite sets. *Discrete Mathematics*, 5:247–251.

Hristidis, V., Gravano, L., and Papakonstantinou, Y. (2003). Efficient ir-style keyword search over relational databases. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, pages 850–861. VLDB Endowment.

Hristidis, V. and Papakonstantinou, Y. (2002). Discover: Keyword search in relational databases. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 670–681. VLDB Endowment.

Huang, F., Li, J., Lu, J., Ling, T. W., and Dong, Z. (2015). Pandasearch: A fine-grained academic search engine for research documents. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1408–1411.

INEX (2011). INitiative for the Evaluation of XML Retrieval (INEX).

Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., and Karambelkar, H. (2005). Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 505–516. VLDB Endowment.

Le, T. N. and Ling, T. W. (2016). Survey on keyword search over xml documents. *SIGMOD Record*, 45(3):17–28.

Li, X., Wang, Y.-Y., and Acero, A. (2009). Extracting structured information from user queries with semi-supervised conditional random fields. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 572–579, New York, NY, USA. ACM.

Liu, F., Yu, C., Meng, W., and Chowdhury, A. (2006). Effective keyword search in relational databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 563–574, New York, NY, USA. ACM.

Liu, Z. and Chen, Y. (2011). Processing keyword search on xml: a survey. *World Wide Web*, 14(5):671–707.

Liu, Z. and Cher, Y. (2008). Reasoning and identifying relevant matches for xml keyword search. *Proceedings of the VLDB Endowment*, 1(1):921–932.

Luo, Y., Lin, X., Wang, W., and Zhou, X. (2007a). Spark: Top-k keyword query in relational databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 115–126, New York, NY, USA. ACM.

Luo, Y., Lin, X., Wang, W., and Zhou, X. (2007b). SPARK: top-k keyword query in relational databases.technical report unsw-cse-tr-0708. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of data*, pages 1–18.

Markowetz, A., Yang, Y., and Papadias, D. (2007). Keyword search on relational data streams. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 605–616, New York, NY, USA. ACM.

Mesquita, F., da Silva, A. S., de Moura, E. S., Calado, P., and Laender, A. H. F. (2007). Labrador: Efficiently publishing relational databases on the web by using keyword-based query interfaces. *Information Processing and Management: an International Journal*, 43(4):983–1004.

Nandi, A. and Jagadish, H. V. (2009). Qunits: queried units in database search. *Computing Research Repository*, abs/0909.1765:20.

Ribeiro, B. A. N. and Muntz, R. (1996). A belief network model for ir. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 253–260, New York, NY, USA. ACM.

Sarkas, N., Paparizos, S., and Tsaparas, P. (2010). Structured annotations of web queries. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 771–782, New York, NY, USA. ACM.

Singh, G., Parikh, N., and Sundaresn, N. (2011). User behavior in zero-recall ecommerce queries. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 75–84, New York, NY, USA. ACM.

Sun, C., Chan, C.-Y., and Goenka, A. K. (2007). Multiway slca-based keyword search in xml data. In *Proceedings of the 16th International Conference on World Wide Web*, pages 1043–1052, New York, NY, USA. ACM.

Takahashi, H. and Matsuyama, A. (1980). An approximate solution for the steiner problem in graphs. *Math Japonica*, 24:573–577.

Tian, Z., Lu, J., and Li, D. (2011). *A Survey on XML Keyword Search*, pages 460–471. Springer Berlin Heidelberg, Berlin, Heidelberg.

Vagena, Z., Colby, L., Özcan, F., Balmin, A., and Li, Q. (2007). *On the Effectiveness of Flexible Querying Heuristics for XML Data*, pages 77–91. Springer Berlin Heidelberg, Berlin, Heidelberg.

Vagena, Z. and Moro, M. M. (2008). Semantic Search over XML Document Streams. In *DATAX (International Workshop on Database Technologies for Handling XML Information on the Web)*.

Wu, X. and Theodoratos, D. (2013). A survey on xml streaming evaluation techniques. *The VLDB Journal*, 22(2):177–202.

Xu, Y. and Papakonstantinou, Y. (2008). Efficient lca based keyword search in xml data. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, pages 535–546, New York, NY, USA. ACM.

Yu, J. X., Qin, L., and Chang, L. (2010). Keyword search in databases.

Zaki, M. J. (2000). Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390.

Zhou, R., Liu, C., and Li, J. (2010). Fast elca computation for keyword queries on xml data. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 549–560, New York, NY, USA. ACM.