Efficient Match-Based Candidate Network Generation for Keyword Queries over Relational Databases

Pericles de Oliveira[†,‡], Altigran da Silva[†], Edleno de Moura[†] and Rosiane Rodrigues[†]

[†]Instituto de Computação
Universidade Federal do Amazonas
Manaus, Brazil
{pericles,alti,edleno,rosiane}@icomp.ufam.edu.br

[‡]NOKIA Solutions and Networks
Rio de Janeiro, Brazil
pericles.oliveira@nokia.com

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, the problem of generating CNs efficiently and effectively has received much less attention. This challenging problem consists of automatically locating relations in the database that may contain relevant pieces of information, given a handful of keywords, and determining suitable ways of joining these relations to satisfy the implicit information needs expressed by a user while formulating his/her query. In this paper, we propose a novel approach for generating CNs, wherein the possible matches for the query in the 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 the 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 demands less resources in terms of processing time and memory. These claims are supported by a comprehensive set of experiments that we carried out using several query sets and datasets used in previous related works and whose results we report and analyze here.

1 Introduction

In the last decade, many researchers have proposed methods to enable keyword searches over relational databases. Their goal is to allow naive/informal users to retrieve information without any knowledge about schema details or query languages. Empowering users to search relational databases using keyword is a challenging task. In particular, the information sought often spans multiple tables and tuples, according to the schema design of the underlying database. Thus, systems that process keyword-based queries over relational databases, commonly called *R-KwS* systems, face the challenging task of automatically determining, from a handful of keywords, the pieces of information to be retrieved and how to combine these pieces 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)*, which are networks of joined relations that are used to generate SQL queries whose results provide an answer to the input keyword query. This approach was proposed in DISCOVER [14] and DBXplorer [2], and was later adopted by a number of other systems, such as Efficient [13], SPARK [18], CD [5], Mincost [10], S-KwS [19], KwS-F [4] and CNRank [8]. Systems in this category take advantage of the basic functionality of the underlying RDBMS by producing appropriate SQL join queries to retrieve answers relevant to keyword queries posed by users. Systems in the second category are based on structures called *Data Graphs*, whose nodes represent tuples associated with the keywords they contain and the edges connect these tuples based on referential integrity constraints. In this approach, adopted by a number of systems, including BANKS [1], Bi-directional [16], BLINKS [11] and Effective [17], 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.

In this paper we present a contribution for the systems in the first category. Specifically, we propose a novel approach for generating Candidate Networks. In a nutshell, our approach aims at pruning the exponential number subsets of combinations of relations that often 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) [4, 19].

Interestingly, since its definition [14], the CN generation problem has been poorly studied in the literature. In fact, most of the following works (e.g., [2, 5, 10, 13, 18]), have focused on the problem of CN evaluation, adopting the CN generation algorithm proposed in [14], called *CNGen*, as default. One of the few exceptions is KwS-F [4], 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 of CN generation is that it requires exhaustive exploration of all, and sometimes, the explosive combination of the multiple keywords occurrence in the database and the multiple ways these occurrences can be joined. The traditional algorithm for generating CNs, CNGen [14], works by first locating the subsets of relations in which the keywords of the query occur. Then, the algorithm executes an exhaustive procedure to generate combinations of these subsets in the form of joined trees that may fulfill the input query.

The approach we propose here, called *Match-Based Candidate*

Network Generation, or *MatCNGen*, enumerates the possible ways that the query keywords can be matched in the database beforehand, to generate query answers. Thereafter, for each of these *query matches*, our CN generation algorithm generates a single CN. We argue that this strategy drastically reduces the time required to generate CNs and we present several experimental results to support this claim.

Besides describing MatCNGen, which we consider as our first main contribution, we also present strategies to efficiently implement the same that comprise our second main contribution. These strategies aim to reduce database operations required to generate CNs to a few sequential disk accesses.

We present and analyse the results of a comprehensive set of experiments that we carried out to evaluate the quality of the CNs generated and the performance of the generation process. These results indicate that MatCNGen is able to generate high quality CNs, more efficiently than the traditional CN generation approach. In addition, the experiments revealed that MatCNGen also has a positive impact on the evaluation of CNs. Specifically, as the generation process avoids generating too many combinations of keyword occurrences, a smaller but better set of CNs is generated. As a result, state-of-art CN evaluation algorithms [13, 18] run faster and produce higher quality results. These trends were observed in experiments we carried out with many distinct queries and datasets.

The rest of the paper is organized as follows. Section 2 reviews the problems of generating and evaluating Candidate Networks in R-KwS systems and related literature. Section 3 presents an overview of MatCNGen and its principal steps. In Sections 4, 5, and 6, we discuss the technical details of each of these steps. Section 7 reports the results of experiments that we have conducted with MatCNGen, comparing the results with those obtained with the representative baselines. Finally, Section 8 presents our conclusions and outlines directions for future work.

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 discusses previous related works 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 [14] that we follow in this paper.

2.1 Schema Graph R-KwS Systems

Figure 1 presents the general architecture of a typical R-KwS system based on schema graphs, which we describe below.

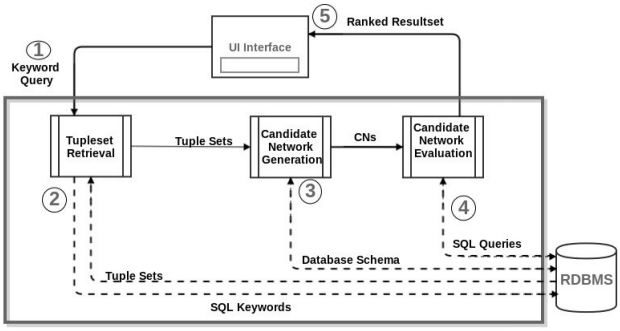


Figure 1: A typical Schema Graph R-KwS System.

Initially, a keyword query is submitted by an user (1). A keyword query searches for interconnected tuples that contain the given keywords. A tuple is said to contain a keyword if a text attribute of the tuple contains the keyword. Given a query, the system looks for subsets of relations that contain the keywords. These subsets of relations, called *tuple-sets*, are then retrieved from the database (2). Then, the tuple-sets are used to generate *Candidate Networks* [2, 14] (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 previously generated , generating CNs also requires information on referential integrity constraints (RIC) of 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 be potentially generated. In practice, however, only a few of them are useful for producing plausible answers [8]. The most well-know algorithm for generating CNs, called *CNGen*, was proposed in [14].

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) [14], that are 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 [2, 14, 19]. In particular, in state-of-the-art systems [5, 13, 18], 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 efficiency and scalability problems in CN evaluation are addressed in a different way in KwS-F [4]. Their approach consists of two steps. First, a limit is imposed on the time that the system spends evaluating the CNs. When this limit is reached, the system must return (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 that the user can select one of the forms, and then the system evaluates the corresponding CN. The authors report a number of experimental results on the effectiveness and feasibility of the proposed approach. Unfortunately, no results about the quality of the results obtained are reported.

To detect the possible duplicate CNs generated by CNGen, Markowetz et. al. [19], proposed ordering the internal nodes of the CNs. This can reduce the number of CNs handled by the systems but still requires that all CNs, whether duplicate or not, are being generated. Our proposed approach avoids duplicated CNs by construction, using the concept of *query match*, which we will properly introduce later.

The important issue of assessing the quality of the answers provided by R-KwS systems has been addressed by Coffman and Weaver [5]. They 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. They compared 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, not a single tested system 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.

2.2 Basic Concepts and Terminology

In the following, we review concepts related to R-KwS systems, following the terminology and definitons introduced in DISCOVER [14]. For convenience, some of the definitions are restated with slight modifications.

Consider a schema graph G representing a relational schema,

where vertices correspond to relations and edges correspond to referential integrity constraints. 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 1. *A **joining network of tuples (JNT)** j is a tree of tuples where for each pair of adjacent tuples t_a, $t_b \in j$, where t_a and t_b are tuples of relations R_a and R_b, respectively, there is an edge $\langle R_a, R_b \rangle$ in G_u and $(t_a \bowtie t_b) \in (R_a \bowtie R_b)$.*

Definition 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.*

Definition 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 4. *Let Q be a keyword query and K be a subset of Q. 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 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 5. *A **joining network of tuple-sets** J is a tree of tuple-sets where for each pair of adjacent tuple-sets R_a^K , R_b^M in J there is an edge $\langle R_a, R_b \rangle$ in G_u.*

Definition 6. *Given a keyword query $Q = \{k_1, \ldots, k_n\}$, a **candidate network** or **CN** 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 6, the answer produced by a CN must be a MTJNT. Totality is enforced by generating CNs that cover all query keywords. For ensuring completeness, a criterion is stablished [14] to determine when the joining networks of tuples produced by a joining network of tuple-sets J may have more than one occurrence of a same tuple. Thus, to be considered as a Candidate Network, any joining network of tuple-sets J must ascertain that this criterion is not fulfilled. Here, we characterize this property by defining which joining network of tuple-sets are considered as *sound*.

Definition 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 using a DBMS to evaluate CNs, we obtain semantically meaningful answers as joined tuples which contain the query keywords.

Example 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 following relational algebra expression:*

$$\sigma_{\text{title} \supseteq \{gangster\}}(\text{MOV}) \bowtie_{\text{id=cid}} \text{CAST} \bowtie_{\text{cid=pid}} \sigma_{\text{name} \supseteq \{denzel,washington\}}(\text{PER}) \quad (1)$$

where MOV *stores information about movies,* PER *stores data about persons (i.e., actors, actresses, directors, etc.) and* CAST *associates persons to the movies in which they work. 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* MOV *and* PER *in Expression 1, are called* non-free tuple-sets*. The remaining operands, such as* CAST *in Expression 1, are called* free tuple-sets*, since they do not contain any of the keywords.*

The complexity of the CN generation is mainly because of 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.

3 MatCNGen Overview

The mains steps of MatCNGen are illustrated in Figure 2. In the following, we present an overview of each of these steps.

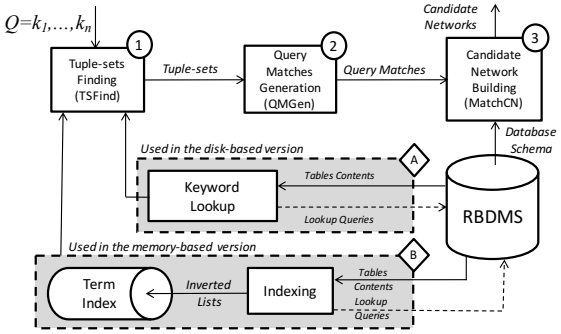


Figure 2: MatCNGen main steps.

Finding Tuple-sets In MatCNGen, when a keyword query is being received for processing, the first step is to identify tuple-sets that can be potentially used in CNs. This corresponds to Step 1 in Figure 2. 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 in 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 the 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 access the disk for each keyword query entered. This strategy is illustrated in Figure 2, in the shaded box labelled "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 that we call the *Term Index*.The Term Index associates each term k from the query to a list, an inverted list to be precise, whose elements are triples of the form $\langle A_i, f_{k,i}, T_{k,i} \rangle$, where A_i identifies an attribute in a relation

[1]Without loss of generality, we assume as in DISCOVER [14], that the attributes in RICs have the same name, and there are no self loops or parallel edges in the G_u. Also, no set of attributes of any relation is both a PK and a FK for two other relations

within 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 in values of A_i. The term index is built in a preprocessing step that scans only once all the relations over which the 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 2 in the shaded box labelled "B". The version of MatCNGen that uses this strategy is called the *memory-based* version, since the term index is stored in memory.

Once the lists of tuples for each keyword is obtained, by one of the strategies presented here, 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 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 MatCNGen, the intersections are computed in memory using an algorithm developed by us based on the ECLAT algorithm [20]. This algorithm that we call 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 the major difference of what is done in all systems based on DISCOVER's CNGen algorithm [14]. 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 details on the TSFind algorithm and the two strategies described above are detailed in Section 6.

Query Matches Generation Given a CN such as the one in Expression 1, we call the set of its non-free tuple-sets a *query match*. As we detail later, matches are the base concept behind MatCNGen, which uses them to build the CNs. The generation of query matches corresponds to Step 2 in Figure 2.

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 match forms a *set cover* of the query.

In our example of Equation 1, the query match is given by:

$$\{\underset{\text{title}\supseteq\{gangster\}}{\sigma}\text{MOV}, \underset{\text{name}\supseteq\{denzel,washington\}}{\sigma}\text{PER}\} \quad (2)$$

In this case, we have the following string covering of the input query: {{*gangster*},{*denzel*,*washington*}}.

Notice that 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 to connect the tuple sets in each query match and build CNs. In Section 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 are spread throughout the database. The set of query matches for such a query must cover all these occurrences. However, only the combinations of keywords that correspond to set covers are considered. We also notice that by imposing the query matches to contain the set covers of the input query, we in fact impose that the matches are minimal and complete. Thus, the CNs assembled using them are also minimal and complete, as required in Definition 6.

Candidate Network Building In Step 3 of Figure 2, 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, our approach for this step consists of building, for each query match M_i, a graph called *match graph*. This graph contains all relations from the original schema graph, as well as the nodes corresponding to the tuples-sets from M_i. These nodes are linked to the original relations according the PK/FK constraints that exist between their base relations and the original relations.

We notice that in [14], 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 built from smaller graphs, the match graphs, as trees that connect *only* the tuple-set forming this graph. Thus, while CNGen requires exhaustive exploration of the full, and sometimes, explosive combination of the all keyword occurrences in the database and the multiple ways these occurrences can be joined, MatCNGen requires exploring several smaller graphs with only a few keyword occurrences. This is done for each query match at a time, drastically reducing the cost of exploring the full graph G_{TS}. Formally, our match graphs can be regarded as subgraphs of the graph G_{TS} induced [9] by query matches.

For building CNs from match graphs we use, as in [14], an adapted version of the well-known breadth-first traversal algorithm. Our strategy of breaking the graph from which the candidate networks are built as smaller graphs leads to a considerable improvement in the time taken to generate CNs. Another interesting effect of this strategy is that we can terminate the graph traversal as soon as the first CN is built. Details on this strategy are discussed in Section 5.

Besides reducing the time for generating CNs, MatCNGen also has 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 [13, 18] run faster and produce higher quality results. These trends were observed across many distinct queries and datasets in the experiments that we performed and report here.

4 Query Matches

In this section we properly introduce the concept of query matches. From now on, we refer to any non-empty subset K of the terms of a query Q as a *termset* of Q.

Definition 8. *Let Q be a query. Let $M = \{R_1^{K_1}, \ldots, R_m^{K_m}\}$ be a set of tuple-sets, where every K_i is a distinct termset of Q. We that M 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 out to be non-total.

In this paper we closely follow the semantics adopted in DISCOVER [14] and only address total and minimal matches. Dealing with other types of matches is left for future work. From now on, we use the term *match* to refer to total and minimal matches.

Example 2. *The following example is based on the sample of the IMDb database made available by Coffman and Weaver [5], whose schema graph is presented in Figure 3*[2].

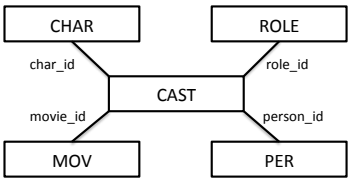


Figure 3: Schema Graph for the IMDb database.

Consider Q={denzel,washington,gangster }, or Q={d,w,g}, for simplicity. This query has 8 minimal covers, among them: C_1={{d},{w},{g}}, C_2={{d,w},{g}}, C_3={{d,g},{d,w}}.

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

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

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 1. *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 of 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 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. □

Obtaining Query Matches

Let Q be a query. The set of all possible termsets of Q is given by $\mathscr{P}^+(Q)$[3]. 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 $K \neq \emptyset$, which 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: $\mathcal{R}_Q = \cup \{\mathcal{R}(K) | K \in \mathscr{P}^+(Q)\}$

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

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

By Definition 8, 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 from the database whose termsets form minimal covers of Q.

$$\mathcal{M}_Q = \{\{R_1^{K_1}, \ldots, R_m^{K_m}\} \in \mathscr{P}^+(\mathcal{R}_Q) \,|\, \{K_1, \ldots, K_m\} \in MC(Q)\} \quad (3)$$

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

At a first glance, the way we state Equation 3 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 [12]. This means that no match for 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 Algorithm 1 we sketch a high-level description of this procedure. It takes as input the keyword query Q and the set $\mathcal{R}_Q$ of non-empty non-free tuple-sets obtained from the database. Our strategy for obtaining $\mathcal{R}_Q$ will be detailed later in Section 6.

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

Algorithm 1: Query Matches Generation

The algorithm generates all subsets of $\mathcal{R}_Q[i]$ with sizes $i = 1, 2, \ldots, |Q|$ (Line 3) of $\mathcal{R}_Q$. Then, in Line 5, the algorithm selects as query matches those subsets of tuple-sets whose keywords form minimal covers of Q.

It is easy to see that Algorithm 1 has a time complexity of $\sum_{i=1}^{|Q|} \binom{|\mathcal{R}_Q|}{i}$. This equation gives us an upper bound on the number of query matches that must be generated for a query. It shows that the running time depends on two important factors: the size of the query and on the size of the sets of tuple-sets $\mathcal{R}_Q$.

Regarding these two factors, the first one, the size of a 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 termsets are distributed among 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 3. *To illustrate how large the set of query matches can be in practice, consider the query* $Q' = \{denzel,washington\}$. *In the sample of the IMDb database we used [5], the set* $\mathcal{R}_{Q'}$ *has* 6 *non-empty tuple-sets. Thus, 21 subsets of* $\mathcal{R}_{Q'}$ *are generated by Algorithm 1, and out of which only five turn to be query matches.*

Now, by adding a single term $\{gangster\}$ *to the query, we have* $Q = \{denzel,washington,gangster\}$ *and* $\mathcal{R}_Q$ *has* 10 *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 Algorithm 1 in practice will be provided when we report our experimental results in Section 7.

5 Generation of Candidate Networks

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 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). We now describe our algorithm to generate CNs by finding these free tuple-sets.

In [14], the generation of CNs is a procedure that extracts JNTs from a graph that represents the possible ways of connecting the tuple-sets of the query. This graph is called *a tuple-set graph*.

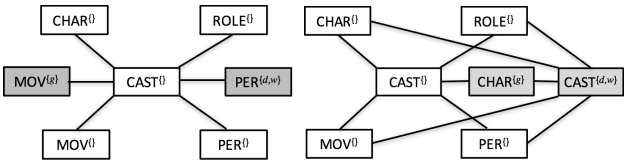
Definition 9. *A **tuple-set graph** G_{TS} for a query Q is a graph whose nodes are all the non-empty tuple-sets $R_i^{K_i}$, where $K_i \subseteq Q$, including the free tuple-sets, and there is an edge $\langle R_i^{K_i}, R_j^{K_j} \rangle$ in G_{TS} if the schema graph G_u has an edge $\langle R_i, R_j \rangle$.*

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 10. *Let G_{TS} be a tuple-set graph for a query Q, let M be a query match of Q and F be the set of free tuple-sets from G_{TS}. 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 10 relies on the fundamental concept of subgraphs induced by subsets of vertices [9] to characterize a match subgraph $G_{TS}[M]$. $G_{TS}[M]$ is, thus, a subgraph of G_{TS} whose nodes consist of the nodes of M, that is, the non-free tuple-sets composing match M, and those of 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. *Considering the same database and query from Example 2, Figure 4 shows two match graphs induced by matches M_2 and M_3, respectively, which are highlighted with shaded nodes. In all these graphs, notice that the free tuple sets correspond to the relations in the schema graph of Figure 3.*



$G_{TS}[M_2]$ $G_{TS}[M_3]$

Figure 4: Example of two match graphs.

Recall from Section 4 that, as our example query has three keywords; any query match will have at most 3 tuple-sets and, thus, any match graph $G_{TS}[M_i]$ will include at most 8 nodes, that is, 3 non-empty non-free tuple-sets plus 5 free tuple-sets.

The MatchCN Algorithm

In Algorithm 2 we describe MatchCN, the algorithm we proposed to generate CNs from match graphs. Like the original CNGen algorithm [14], MatchCN is also based on the well-known breadth-first traversal algorithm. However, other algorithms for extracting meaningful trees from graphs can also be used [9]. Exploring such algorithms for this task is left as future work.

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

Algorithm 2: MatchCN Algorithm

SingleCN(M,$G_{TS}[M]$)
Input: A query match M, a match graph $G_{TS}[M]$
Output: A single CN C
1: $\mathcal{F} \leftarrow \emptyset$ {Initialize a queue}
2: $J \leftarrow \{R_1^{K_1}\}$ from M
3: Enqueue($\mathcal{F}$,J)
4: **while** $\mathcal{F}$ not empty **do**
5: $J \leftarrow$ Dequeue($\mathcal{F}$)
6: **for each** $R_v^{K_v}$ in $G_{TS}[M]$ adjacent to some $R_u^{K_u}$ in J **do**
7: **if** $R_v^{K_v}$ is a free tuple-set **or** $R_v^{K_v} \notin J$ **then**
8: $J' \leftarrow J$
9: Expand J' with tuple-set $R_v^{K_v}$ joined to $R_u^{K_u}$
10: **if** $J' \notin \mathcal{F}$ **and** $|J'| \leq T_{max}$ **and** J' is *sound* **then**
11: **if** J' contains the match M **then**
12: **return** J' {*Return current JNT as a valid CN*}
13: **else**
14: Enqueue($\mathcal{F}$,J')
15: **end if**
16: **end if**
17: **end if**
18: **end for**
19: **end while**
20: **return** {}

Algorithm 3: SingleCN Algorithm

The algorithm takes as input a set of query matches generated by Algorithm 1 and a tuple-set graph as in Definition 9. In the Loop 2–6, the algorithm processes each query match from $\mathcal{M}$ to generate one candidate network corresponding to it. Given a query match M, in Line 3, the match graph induced from the tuple-set graph is first generated. Then, in Line 4, the algorithm calls a procedure to generate a single CN from this match graph. This procedure, named *SingleCN* will be detailed next. Finally, the generated CN C is added to set of CNs for the query (Line 5).

The SingleCN procedure is described in Algorithm 3. Given a query match M and match graph $G_{TS}[M]$, it generates the Candidate Network as the shortest sound JNT that contains the match, if at least one Candidate Networks of size up to T_{max} exists, where T_{max} is global threshold that controls the number of tuple-sets in a given CN. Thus, notice that at most one CN is generated for each query match.

The procedure performs a breath-first traversal over the match graph, starting from the node representing the first tuple-set in the match (Line 1). The Loop 4–19 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 9).

To be considered to form a valid CN, a JNT J' generated in the loop must satisfy three conditions (Line 10): (1) it must not have been generated previously ($J' \notin \mathcal{F}$) ; (2) its size, that is, 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 later mitigated in [19]. 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 find any case in which this condition appeared. Condition (3) ensures that the joining networks of tuples produced by J' do not

have more than one occurrence of a tuple, according to Definition 7.

If the current JNT satisfies these conditions, the algorithm verifies whether it contains all tuple-sets from the input query match (Line 11). In this case, this JNT is returned as a valid CN (Line 12) and the generation process terminates. Otherwise, the current JNT is enqueued (Line 14) to be further expanded in Line 9.

If during the process none of the generated JNTs satisfy the conditions above, the queue will eventually become empty, and the process terminates with no CN generated for the current match (Line 20).

Example 5. *Table 1 illustrates execution of Algorithm 3 while taking as input the match graph $G_{TS}[M_3]$ of Figure 4. Each group of lines in the table refers to an iteration of Loop 4–19, numbered in Column* I#*, except for the 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 2) and enqueued (Line 3). 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 10 and 11, and is thus returned as a valid CN.*

I#	Queue	Operations
0	$\mathsf{MOV}^{\{g\}}$	generate (L.2), enqueue (L.3)
1	$\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}}$	expand (L.9), enqueue (L.14)
2	$\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}} \bowtie \mathsf{CHAR}^{\{\}}$	expand (L.9), enqueue (L.14)
	$\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}} \bowtie \mathsf{ROLE}^{\{\}}$	expand (L.9), enqueue (L.14)
	$\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}} \bowtie \mathsf{MOV}^{\{\}}$	expand (L.9), enqueue (L.14)
	$\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}} \bowtie \mathsf{PER}^{\{\}}$	expand (L.9), enqueue (L.14)
	$\mathsf{MOV}^{\{g\}} \bowtie \mathsf{CAST}^{\{\}} \bowtie \mathsf{PER}^{\{d,w\}}$	expand (L.9), return CN (L.12)

Table 1: Execution of Algorithm 3 for $G_{TS}[M_3]$ of Figure 4.

MatchCN and CNGen

As we have already noted, MatchCN and the original CN generation algorithm, CNGen, are both based on a breadth-first graph traversal procedure. However, these two 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, that is the match graphs. By doing so, MatchCN avoids exploring non-relevant paths in the full tuple-set graph. For instance, considering our current example, in the tuple-set graph for the query Q the node corresponding to $\mathsf{CAST}^{\{\}}$ is adjacent to every tuple-set from the other relations 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 resource consumption reported in the literature. In MatchCN, each graph traversal finishes as soon as a valid CN is found. 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 shown in the results of experiments that we report in this paper.

6 Finding Tuple-sets

As shown in Algorithm 1, a necessary step for generating query matches is to find the non-empty, non-free tuple-sets corresponding to the input query, that is, the tuple-sets that effectively contain some termset of the query. In this section, we present our approach to accomplish this task.

A straightforward approach to obtain tuple-sets is to scan every table from the database and look for the occurrence of every termset of the input query. This would require looking for all possible termsets of the query in each tuple in the database. To avoid this, DISCOVER [14] 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 the 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.

TSFind Algorithm

We propose the TSFind algorithm (Algorithm 4), which is based on the ECLAT [20] algorithm for finding frequent itemsets. Following the notation introduced in Section 4, our algorithm aims at building the set $\mathcal{R}_Q$ of existing tuple-sets for a given keyword query Q in a database instance.

```
   TSFind (Q)
   Input: A keyword query Q={k_1, k_2, ..., k_m}
   Output: Set of non-free and non-empty tuple-sets R_Q
 1: {Part 1: Find sets of tuples containing each keyword}
 2: let D the current instance of the target DB
 3: P ← ∅
 4: for each keyword k_i ∈ Q do
 5:    add ⟨{k_i}, ∅⟩ to P
 6:    for each relation R ∈ D do
 7:       {Issue queries over R looking for tuples containing k_i}
 8:       T_{k_i} ← set of tuples in R containing k_i
 9:       update ⟨{k_i}, T_{k_i}⟩ in P
10:    end for
11: end for
12: {Part 2: Find sets of tuples containing larger termsets}
13: P ← TSInter(P);
14: {Part 3: Build tuple-sets}
15: R_Q ← ∅
16: for each ⟨K, T_K⟩ ∈ P do
17:    R_Q ← R_Q ∪ {R^{K} | there is some tuple of R in T_K}
18: end for
19: return R_Q
```

Algorithm 4: TSFind Algorithm

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 parts. In the first part, the algorithm finds sets of tuples that contain each keyword of the query. For this, in the Loop 4–11 the algorithm iterates over each keyword k_i in the query and issues a SQL query over each relation R (Loop 6–10) 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 9). When the loop finishes (Line 11), all pairs in $\mathcal{P}$ refer only to singleton termsets.

In the second part, the algorithm invokes in Line 13 a recursive procedure called TSInter to find sets of tuples containing larger termsets. This procedure will be detailed in a moment.

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 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.

The recursive algorithm TSInter is presented in Algorithm 5. 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 5. 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.

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

Algorithm 5: TSInter Algorithm

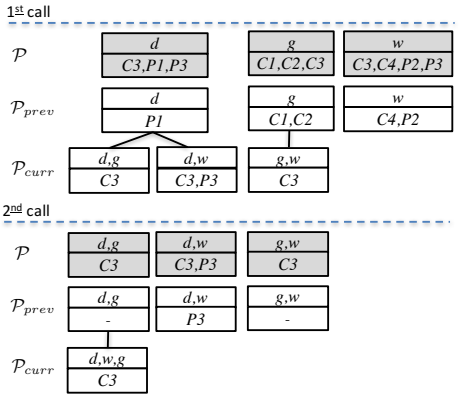


Figure 5: Finding non-free, non-empty subsets.

In the algorithm, the Loop 3–12 combines pairs of termsets in $\mathcal{P}$. The termsets are numbered and processed in numerical order, only to avoid processing the same pair of tuple-sets twice. In Line 5, 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 8). 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 9 and 10), whose role is to keep track of the new state of the input $\mathcal{P}$.

In our example of Figure 5, 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 $\mathcal{P}_{prev}$ to reflect $\langle \{d\}, \{P1\}\rangle$ and $\langle \{w\}, \{C4, P2\}\rangle$. Similarly, the second call finds the sets of tuples containing termsets with three keywords.

The algorithm makes a recursive call in Line15 to look for sets of tuples that may include larger termsets. Each call of TSInter returns the new state of $\mathcal{P}$, which is represented by $\mathcal{P}_{prev}$, and expanded with sets of tuples containing larger termsets, which are stored in $\mathcal{P}_{cur}$(Line 17).

Using In Memory Index

As we have discussed, Algorithm 4 is likely to save on 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 to scan 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 database accesses 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 Algorithm 6. It is very similar to Algorithm 4. In fact, the only diference from **TSFind** is in 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.

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

Algorithm 6: 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 that we carried out and the results are reported in this paper. 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.

7 Experiments

In this section, we report a comprehensive set of experiments performed using several databases and query sets previously used in similar experiments from related work. Our goal is to analyze two main aspects. First, the quality of the set of CNs generate, that is, how the CNs generated by MatCNGen impact the results obtained by well-know CN evaluation algorithms. Second, the performance and scalability of the algorithms we propose.

7.1 Experimental Setup

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.

Besides MatCNGen, 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 [14], Efficient [13], Bi-directional [16], Effective [17], SPARK [18] and CD [5], as well as systems based on the Data Graph approach, such as BANKS [1], Min-cost [10] and BLINKS [11].

As a baseline, we used the implementation of CNGen [14] on the Efficient system [13]. 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 [5, 13, 14, 17, 18]. The implementation used here was gently provided by its authors. As this implementation was originally targeted to the ORACLE DBMS, we had to adapt it to work with PostgresSQL, which we used in our implementations. Thus, all tested implementations ran under exactly the same operational conditions.

We used five datasets: IMDb, Mondial, Wikipedia, DBLP and TPC-H, which were previously used for the experiments reported in previous work [5, 18]. In Table 2 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[4]. We use the TPC-H database with size of 876MB instead of 100MB as used in previous work [14].

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 were reported by Coffman and Weaver [5]. There were 42 to 45 queries targeted to the IMDb, Mondial and Wikipedia.

[4]Further details on IMDb, Mondial and Wikipedia datasets are provided in [6]. More details on the DBLP dataset are given in [7]

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 2: Characteristics of the datasets used.

SPARK: Queries used in the experiments were reported for SPARK [18]. 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 used in the experiments by Coffman and Weaver [5]. We then replaced these queries with equivalent queries that mention names of persons instead. The remaining fifteen queries were used without modification. For the DBLP dataset, two out of 18 queries were replaced because they did not have results in the database. For the Mondial dataset, all 35 original queries were used.

INEX: Fourteen queries specified for the INEX 2011 challenge [15] 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 is presented in Table 3. The complete list of all queries used is available in `http://www.a.b.c`[5]

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 3: Overview of the experimental query sets.

An important parameter that impacts CN generation is the number of keywords used in the queries. In Table 4, we present the maximum and the average number of keywords used in the queries of the query sets in our experiments. Considering all query sets, the average number of keywords per query is 2.1 and the maximum is four. These numbers are indeed typical 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 4: 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 4.

7.2 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 [4], our goal here is to provide a perspective of the scalability of our method. These numbers are useful for validating the assumptions we made while designing our method and to explain the performance results we achieved.

[5]The URL will be available upon the publication of the paper.

Table 5 presents the maximum and the average number of query matches generated for each pair of query set/dataset. The number of query matches is 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 2). Still, the average number of matches for the 218 queries that we used is lower than 17.

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 5: Number of query matches generated.

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 includes more than 100 Referential Integrity Constraints (see Table 2). This increases the number of ways tuple-sets can be combined to generate query matches.

In Figure 6, we compare the average number of CNs generated by MatCNGen and CNGen. The implementation of CNGen [14] we used is available as part of the Efficient system [13], which was kindly made available by its authors.

Notice that MatCNGen 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 MatCNGen generated less than 10% of the CNs generated by CNGen. The fact that MatCNGen generates less CNs than CNGen for all query sets with all databases was expected, since at most a single CN is generated for each query match. Thus, the number of CNs cannot be higher than 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 7.4. Interestingly, as we will discuss in detail in Section 7.3, 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.

7.3 Quality Results

In this section we study about the impacts of our method on the quality of the results of processing keyword queries. Our evaluation consists of measuring the quality of the output produced by the Hybrid [13] and the Skyline Sweeping [18] algorithm, while taking as input the set of CNs generated by our method, MatCNGen. Both are well-known algorithms used for evaluating a set of CNs over a database, providing a set of joining networks of tuples (JNT) as result. The implementations used here were kindly provided by their respective authors. We report results obtained with two configurations: MatCNGen with Hybrid (MCG+H) and MatCNGen with Skyline Sweeping (MCG+SS).

Quality Metrics

To evaluate the results produced by each system/configuration for each query set, we used the well-known *Mean Average Precision (MAP)* metric [3]. 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 one if the answer at position k is relevant or zero 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 [3], 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. The MRR value is obtained for all queries in a query set by taking 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.

Results – Coffman-Weaver Query Set

In their paper [5], Coffman and Weaver reported results of quality experiments carried out with queries from the Coffman-Weaver Query Set using several R-KwS systems previously presented in the literature: BANKS [1], DISCOVER [14], Efficient [13], Bidirectional [16], Effective [17], DPBF [10], BLINKS [11], SPARK [18] and CD [5]. The authors of cordially provided us with these results, along with the set of relevant tuples that should be returned by each query, which were used as the golden standard to measure the quality of the results obtained by each system. These resources allowed us to compare our two configurations, MCG+H and MCG+SS, with these systems.

The overall result for all queries of the Coffman-Weaver query set is presented in Figure 7 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 while processing these queries are presented in Figure 8.

In this 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.

Results – Spark and INEX Query Sets

In case of the queries from 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 available query descriptions. Then, we ran quality experiments using configurations obtained by coupling CNGen with Hybrid and with Skyline Sweeping as baselines. For sake of symmetry, we named these configurations CNGen+H and CNGen+SS, respectively. Notice, however, that they in fact correspond to systems DISCOVER [14] and Efficient [13], respectively.

The results are presented in Figure 9. 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 achieved only for queries that return a single JNT as result. There were 63 such queries for the SPARK query set and eleven for the INEX query set.

In these 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+H outperformed MCG+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

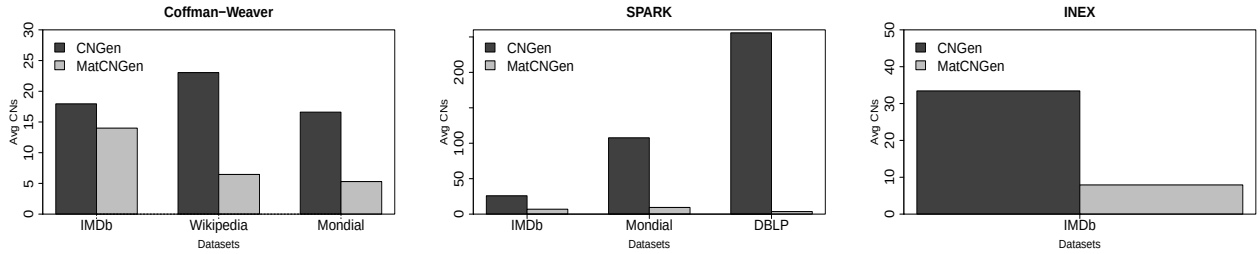


Figure 6: Average number of CNs generated for all query sets and datasets.

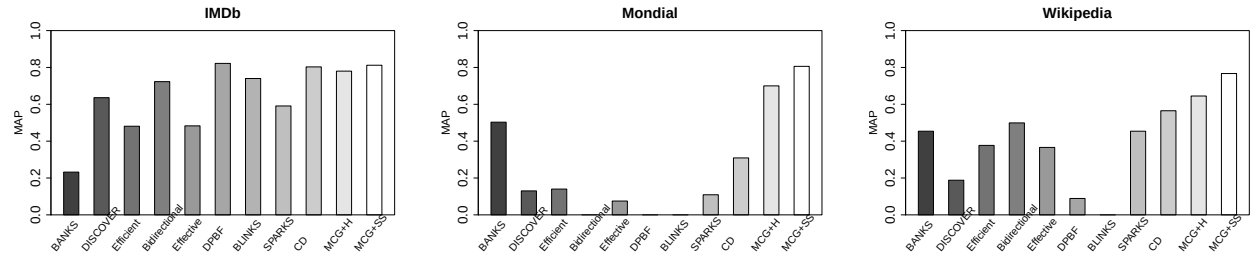


Figure 7: MAP measured across the various systems and datasets for queries from Coffman-Weaver.

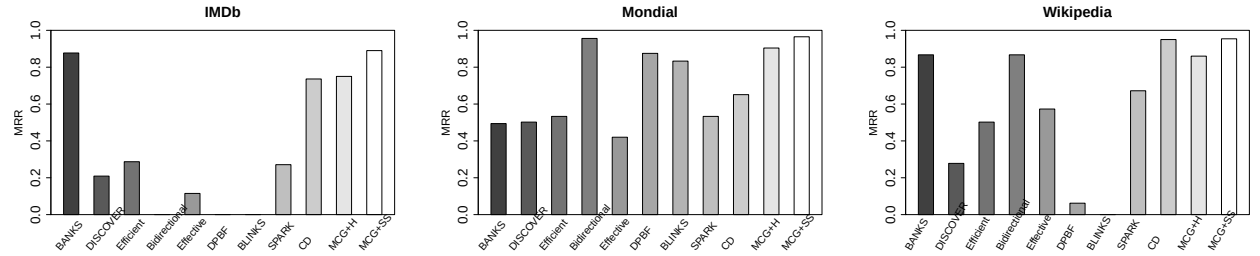


Figure 8: MRR results for each system for queries from Coffman-Weaver where exactly one JNT is relevant.

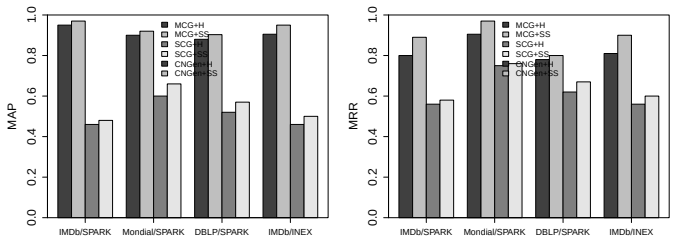


Figure 9: MRR and MAP for queries from SPARK and INEX.

all tests we performed. This corroborates with our claims regarding the quality of the CNs generated and their potential of positive impact in the results produced by RwK-S systems.

7.4 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. For these experiments, we used two different versions of MatCNGen: *MatCNGen-Disk*, in which the generation of tuples is carried out using the disk-based version of the TSFind algorithm (Algorithm 4); and *MatCNGen-Mem*, which uses the counterpart memory-based version (Algorithm 6).

Overall Results

Figure 10 compares the average time spent for generating the corresponding Candidate Networks for all queries in each query set 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 taken 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 (Section 5). Notice that in the case of MatchCN, the time also includes the generation of query matches.

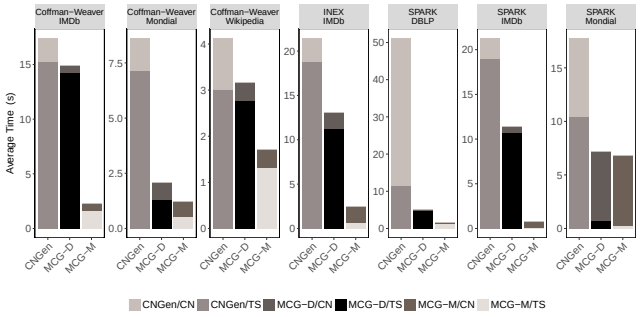


Figure 10: Time to generate CNs using CNGen and MatCNGen.

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 faster than both by far. As expected, this is more advantageous while dealing with datasets that have many relations, i.e. IMDb, Wikipedia and DBLP, since potentially more tuple sets would be generated for each query.

Regarding the time to assemble CNs, as expected, the fact that the MatchCN algorithm builds a single CN for each query match resulted in a time improvement in all cases. This impact is higher

in cases such as SPARK/DBLP, which has a few query matches (see Table 5), than in cases having many query matches, such as SPARK/Mondial.

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 [4, 19], since queries with a high number of keywords usually cause excessive memory consumption.

In the experiments we reported so far, we have not faced this issue, since, as shown in Table 5, 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, explained next. 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 11. In Figure 11 (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 11 (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 11 (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 of KwS-F [4]. 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 11 (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 keyword are very unlikely and that very often queries have two or one keywords.

Discussion

Previous works in the literature have reported that processing Keyword Queries over relational databases has unpredictable running times, with certain queries taking too long or even failing due to memory exhaustion [4] [5]. Indeed, the generation of Candidate Networks can be a costly operation regardless of the method used. For instance, on an 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 50 seconds, on average, to generate the CNs for the queries over the DBLP dataset, which is too slow for on-line applications. On the other hand, our MatCNGen memory version took about 0.5 second 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 7.3.

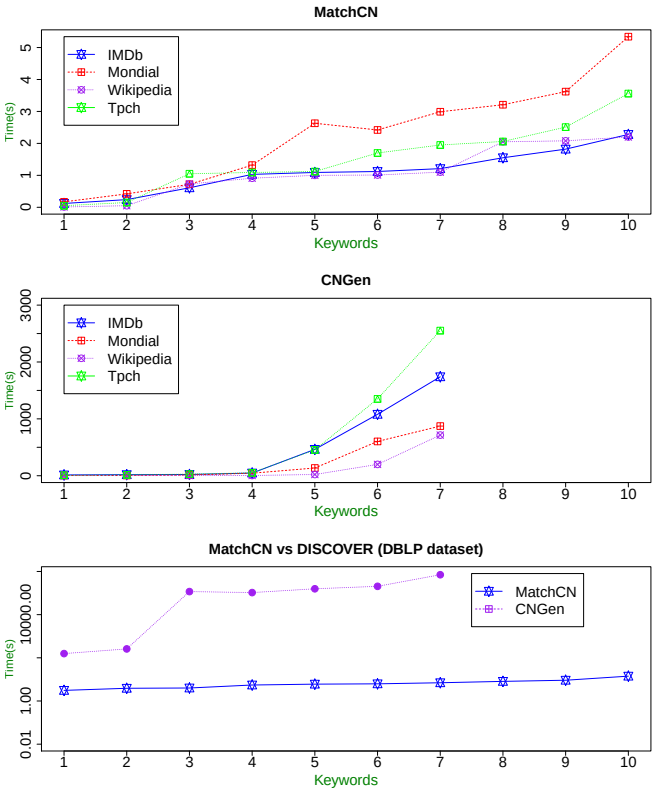


Figure 11: Time to generate CNs with varying number of keywords.

8 Conclusion

In this paper we presented a novel approach for generating Candidate Networks (CN), which are relational joined expressions that compose the core of several R-KwS. Our approach, *Match-Based 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. These claims are supported by a comprehensive set of experiments that we carried out using several previously used query sets and datasets, which were also used in the related works in the past. In particular, we have shown that our approach makes it viable for R-KwS system to process keyword queries in an on-line setting. So far, the previous works in the literature have reported that this task had unpredictable running times, with certain queries taking too long to run or even failing due to memory exhaustion.

Our experience in the development of the MatCNGen has raised a number of ideas for future work. First of all, for building CNs from match graphs we have used a breadth-first traversal algorithm. However, there are alternatives in the literature that we intend to explore and compare in the near future [9]. Regarding the process of finding tuple-set, we plan to devise suitable mechanisms to keep the in-memory term index up-to-date with possible changes in the database, e.g., by using triggers. In addition, we plan to address the problem of better memory management by using compression techniques or partial use of secondary memory in case it is needed. Finally, we are also interested in adapting the CN generation and evaluation mechanisms to be carried out inside the DBMS engine, motivated by the idea of making the process even more scalable and easier to deploy in real applications.

9 References

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

[2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proceedings of the 18th IEEE International Conference on Data Engineering*, pages 5–17, 2002.

[3] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 2nd edition, 2011.

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

[5] J. Coffman and A. C. Weaver. A framework for evaluating database keyword search strategies. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, pages 729–738, 2010.

[6] J. Coffman and A. C. Weaver. Relational keyword search benchmark, 2010. http://www.cs.virginia.edu/ jmc7tp/resources.php.

[7] R. Cyganiak. Benchmarking D2RQ v0.1. DBLP dataset, 2016.

[8] P. de Oliveira, A. da Silva, and E. de Moura. Ranking Candidate Networks of relations to improve keyword search over relational databases. In *Proceedings of the 31st IEEE International Conference on Data Engineering*, pages 399–410, 2015.

[9] R. Diestel. *Graph Theory*. Springer, 4th edition, 2012.

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

[11] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: Ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 305–316, 2007.

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

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

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

[15] INEX. INitiative for the Evaluation of XML Retrieval (INEX), 2011.

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

[17] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 563–574, 2006.

[18] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: Top-k keyword query in relational databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 115–126, 2007.

[19] A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 605–616, 2007.

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