POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Query Depth in Relational Databases: Ensuring the Legality of Searches on Business Records Metadata

Relatore: Prof. Davide MARTINENGHI

Tesi di Laurea di:
Luca GRAZIANI, matricola 801232

Anno Accademico 2014-2015

To my family.
To my friends who were always there.

ACKNOWLEDGEMENTS

I would like to thank my advisor prof. Davide Martinenghi for his support during the last months. He has always offered valuable insights and suggestions on this work but he also let me work independently.

I would also like to thank prof. Lenore D. Zuck from the University of Illinois at Chicago for her support and trust. This whole research has originated from her and I wouldn't be writing this if it wasn't for her.

I also thank prof. Marco Santambrogio who gave me advice on how to live my double degree experience from the start.

I would also like to thank all the great professors and mentors who inspired me over my entire study career at Politecnico.

Thank you to all my colleagues and friends that I met during the last 6 years and with whom, although studying hard, I have had so much fun.

A special thanks goes to Emanuele Del Sozzo, Enrico Deiana and Marco Rabozzi, along with all the *UIC-PoliMi* students of the 2013 class, for their support and friendship.

LG

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF FIGURES

LIST OF TABLES

TABLE **PAGE**

ABSTRACT

Intelligence agencies worldwide have access to databases containing daily business records metadata, such as telephony metadata, with the aim of preventing and identifying terrorist activity. In 2013, the US Foreign Intelligence Surveillance Court (FISC) released a number of documents on the use of telephony metadata by the US National Security Agency (NSA). NSA claimed that they did not possess technical expertise on how to correctly conduct searches on the BR metadata. These documents revealed that the NSA was querying a larger set of identifiers than the one permitted in previous FISC orders. This motivates the need for an automated tool with the capability of analyzing the legality of queries on telephony metadata. This analysis would help to prevent the execution of queries which retrieve more identifiers than allowed.

The aim of this thesis is to conceive a tool for such purpose, which, differently from the previous work on the subject, will conduct an analysis based only on the query itself and not on the output that the query produces on a specific database. The input queries will be checked against the guidelines specified in a 2013 Obama administration white paper on bulk collection of telephony metadata. To achieve this goal we propose two different notions of *depth* of a query: the *graph-depth* approach is a straightforward verification of the above guidelines while the *cost-depth* approach, although comparable to the previous one, is inspired by the theory of *databases with access limitations* and solves a more complex, although similar, problem. We will propose a number of algorithms for both approaches and we will implement them as part of the *QueryAnalyzer* system. Finally, we will compare the performances of the different algorithms.

SOMMARIO

Le agenzie di sicurezza governativa fanno spesso accesso ai metadati delle chiamate telefoniche, tra cui mittente e destinatario, allo scopo di identificare e prevenire attività terroristiche. Nel 2013, la Corte di Vigilanza sull'Intelligence Estera americana (FISC) ha rilasciato dei documenti riguardanti l'uso dei metadati telefonici da parte dell' Agenzia di Sicurezza Nazionale americana (NSA). Questi documenti hanno rivelato che l'NSA ha condotto ricerche accedendo ad un numero di identificatori di numeri telefonici maggiore rispetto a quanto in precedenza stabilito dalla FISC. Un documento governativo, redatto dall'amministrazione Obama nel 2013, stabilisce che l'NSA è abilitata a interrogare i metadati telefonici partendo da una "lista nera" di id possedenti la qualifica di "*reasonable articulable suspicion*" (RAS) e accedendo a id distanti non più di tre *passi* (hop) dagli id RAS, col significato che due id sono distanti di un passo quando tra essi è intercorsa una chiamata, due passi se sono connessi solo indirettamente attraverso una terza parte e così via. L'NSA ha risposto alle accuse della FISC dichiarando che "dal punto di vista tecnico, nessuno aveva una completa conoscenza del sistema di interrogazione dei metadati telefonici". Lavori precedenti sul problema di prevenire l'accesso illegale dell'NSA ai suddetti metadati ha dimostrato che non solo è possibile filtrare il database nascondendo gli id distanti più di tre passi dagli id RAS, ma tale operazione è anche efficiente se si utilizza un *database a grafi*.

Lo scopo della nostra ricerca è di quantificare il massimo numero di passi attraversati dalla query andando ad analizzare la struttura della query e non l'output prodotto dalla sua esecuzione. A questo scopo abbiamo definito uno schema di database relazionale e un sottoinsieme di SQL atti a rappresentare le query di interesse. Abbiamo quindi definito la *graph-depth* di una query come il massimo numero di passi espansi dalla query. Ad esempio, consideriamo la seguente query (espressa in linguaggio naturale) "*Trova Alice, Bob e Carl tali che Alice ha chiamato qualche id RAS che ha chiamato Bob che, a sua volta, ha chiamato Carl*": in generale Alice e Bob non sono id RAS ma sono in diretto contatto con qualche RAS, per cui entrambi sono lontani un passo dagli id RAS. Carl invece, è in contatto con qualche RAS tramite Bob per cui esso dista due passi dalla lista nera. Da questo segue che il massimo numero di passi espansi dalla query è due. Siamo stati in grado di sviluppare un algoritmo che, in tempo polinomiale, calcola la graph-depth di una query, purchè quest'ultima sia minima.

Il passo successivo della nostra ricerca è stato modificare leggermente il problema per considerare la direzione delle chiamate. Infatti la definizione di passo definita dai documenti legislativi statunitensi è informale e non specifica se il verso delle chiamate influenzi o meno la legalità della query. Per questo abbiamo sviluppato un concetto alternativo di profondità chiamato *cost-depth*. L'idea è basata sulla nozione di *accesso*, ovvero l'azione di fornire un elenco di id in input al database per estrarre o coloro che hanno ricevuto

chiamate da tale elenco o coloro che hanno effettuato chiamate verso tale elenco. Questo concetto permette di distinguere la direzione delle chiamate. La cost-depth di una query è quindi il minimo numero di accessi richiesti per eseguire la query. Consideriamo nuovamente la query sopra che per praticità riscriviamo come "*trova A, B e C tali che A chiama RAS che chiama B che chiama C*". Per eseguire un accesso occorre avere una lista nota di id, che nel nostro caso sono proprio i RAS. Dai RAS possiamo risalire a chi li ha chiamati ottenendo gli A (primo accesso). Usando gli stessi RAS risaliamo a chi hanno chiamato ottenendo B (secondo accesso). Ora i B sono noti e possiamo risalire a chi hanno chiamato ottenendo C (terzo accesso). Il primo e secondo accesso sono diversi perchè diversa è la direzione in cui andiamo ad estrarre i dati e sono entrambi diversi dal terzo che usa una lista di input diversa. La cost-depth della query è quindi tre. L'esempio potrebbe indurre il lettore a pensare che la cost-depth sia semplicemente il numero di chiamate che intercorrono tra le varie parti, ma così non è: nella query "*trova A tale che RAS chiama RAS che chiama A*" sono richieste due chiamate che connettono le varie parti ma la cost-depth è in realtà uno. Infatti, estraendo coloro che sono stati chiamati da qualche RAS otteniamo delle coppie $\langle RAS, X \rangle$ (primo accesso), e per trovare i valori per A possiamo riutilizzare le coppie estratte dall'accesso precedente, essendo A stato chiamato da qualche RAS.

In generale ci sono vari piani di esecuzione possibili per una query. Abbiamo inoltre dimostrato che il problema di trovare il piano minimo è NP-difficile il che ci ha spinto a sviluppare un algoritmo esatto basato sul paradigma *branch & bound* e un algoritmo approssimato basato sul paradigma *greedy*. Abbiamo quindi realizzato il sistema *Query-Analyzer* che consente di calcolare la graph-depth e la cost-depth di una query e abbiamo confrontato le prestazioni dei diversi algoritmi.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Intelligence agencies are benefiting from the rapid growth of computational power of computing devices and the constant increase of the capacity of digital storage systems. The storage of *Business Records* (BR) metadata in databases allows intelligence agencies to analyze millions of records with the aim of discovering and preventing the insurgence of threats to national security, such as terrorist activity. However, the use of this data by intelligence agencies is usually constrained in order not to violate citizen privacy. Even though governments do publish guidelines explaining which uses of BR metadata are compliant with privacy laws, these guidelines are not always respected: in 2013 the US National Security Agency's (NSA's) use of bulk *telephony metadata* [1] was detailed in a number of declassified and released Foreign Intelligence Surveillance Court (FISC) documents. A March 2, 2009 order from the FISC [1] addressed the problem of privacy violation: the order revealed that the NSA was querying all the identifiers on an NSA alert list against BR metadata that it received daily. This proved to violate previous FISC orders since the NSA was allowed to query only identifiers for which there was a "reasonable articulable suspicion" (RAS) that the identifier was significant and related to terrorist activity. In January 2009 the NSA's alert list contained 17,835 foreign and domestic identifiers while only about 2,000 identifiers had been given the status of RAS. The NSA justified its actions by stating that "from a technical standpoint, there was no single person who had a complete technical understanding of the BR FISA[2] system architecture. [The court stated to be] exceptionally concerned about what appears to be a flagrant violation of its order in this matter" [1].

The previous example shows that intelligence agencies would greatly benefit from an automated system with the aim of identifying (and possibly rejecting) queries not complying with previously stated orders and guidelines. This system would prevent the intelligence agency analyst to perform illegal queries and therefore would ensure the agency's compliance with the above regulations. In the specific case of the NSA, a 2013 Obama administration white paper [2] provided requirements of legal queries: the white paper claimed that the NSA was allowed to search in the BR metadata up to a distance of three *hops* from RAS identifiers. The concept of hop is informally defined in the white paper: an

[1] Records include information on sender, receiver, originating device, time of conversation, call duration and trunk number.

[2] Foreign Intelligence Surveillance Act

identifier lies at one hop from the RAS list if it is in direct contact with some RAS identifier, i.e. both identifiers take part in the same telephone call, it lies at two hops from the RAS list if it is in direct contact with some one-hop-identifier, etc. So far it is not clear if this requirement held back in the 2008 time frame, the period concerning the order in [1].

1.2 Aim of the thesis

The problem to ensure the legality of queries on telephony metadata according to the guidelines in [2] has been previously faced in two ways: in 2013, *Kanich et al.* reasonably assumed that telephone data were being stored in a relational-like database and proposed an input driven approach towards the problem. The approach involved filtering the database to remove the non legally obtainable identifiers [3]; In 2014, *Panebianco* reproduced the NSA's query process using *graph databases* to compare the obtained performance to those obtained with traditional SQL relational databases [4].

The purpose of this work is to devise an automated system capable of deciding an input query's compliance to some requirements. The considered data will still be telephony data, similar in structure to those queried by the NSA, since it is reasonable to assume that most intelligence agencies worldwide use it in their investigations[1]. As far as technology is concerned, we will assume the data is stored in an SQL relational database system, since it still is the most common type of database technology to date [6].

We will devise a first approach, named *graph-depth*, to ensure the compliance of queries with respect to the guidelines in the white paper [2] by computing the maximum number of hops expanded by the query. For instance, a query of the type "*find all the Alice, Bob and Carl s.t. Alice called some RAS id who called Bob who called Carl*" has a graph-depth of two, because in the worst case Alice, Bob and Carl are not RAS, and therefore Bob and Alice lie one hop apart from the RAS id's while Carl, being in contact with Bob, lies at two hops from RAS and therefore determines the graph-depth of the query. Even though such guidelines specifically refer to the NSA, it is reasonable to assume that governments worldwide have defined similar guidelines on the matter.

We will then propose a second approach, named *cost-depth*, which ensures that queries only perform a certain number of operations, called *accesses*, on the telephony metadata. An access is the action to provide a set of known id's to the database to extract those who called them or those who were called by them. A query is then labeled as legal or illegal based on how many accesses it requires. For instance, in order to access the query above, we require one access to extract Alice by looking at the callers of RAS id's, which are the only known id's at the beginning, one access to extract Bob by looking at the callees of said RAS id's and one access to extract the receivers of Bob's calls (Carl). Because of this the cost-depth of the query is three, which is greater than its graph-depth. This should not

[1] Even though the NSA's bulk telephony metadata program is being discontinued, telephony data will still be used by the NSA for intelligence purposes [5].

convince the reader that the cost-depth of a query is simply the number of calls connecting the various parties involved, for instance the query "*find all the Alice s.t. some RAS called some RAS who called Alice*" has a cost-depth of one, because it suffices to extract the id's X that have been contacted by some RAS id and select only those X that are RAS which have called somebody as well. Alice is the id called by the filtered X. This operation requires only one access.

We will then develop a program named *QueryAnalyzer* which, given an input query, implements the two approaches. The peculiarity of both approaches is that the metrics they compute, named *graph-depth* and *cost-depth* respectively, are derived by analyzing the syntactic structure of the query and *not* by looking at the results of its execution as done in [3]. Should the computation reveal that the query is not legal, the security analyst will then decide how to address the issue. Two possibilities are to simply discard the query, and coming up wuth another one, or to execute it on a filtered database as proposed in [3]. The supposed work flow of an intelligence analyst querying the telephony metadata is shown in Figure 1, while the work flow modified by the addition of *QueryAnalyzer* is shown in Figure 2.

Figure 1: Example of intelligence analyst work flow

1.3 Contributions

This thesis contributes to the research in [3, 4] by presenting two approaches toward enforcing the legality of queries on telephony metadata. Here are the novel features of our study:

- the proposed metrics of *graph-depth* and *cost-depth* depend only on the query and not on the output produced from a specific database. Therefore they perform a conservative assessment of the depth of the query;
- the definition of *cost-depth* comes from the theory of databases with access limitations [7]: we use the concepts of *input* and *output* attributes of a relation schema to develop a novel definition of *access* used to quantify the amount of information

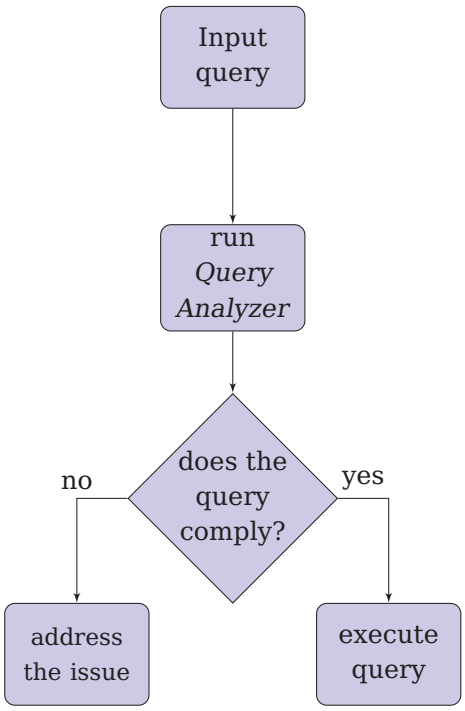


Figure 2: Example of intelligence analyst work flow using *QueryAnalyzer*

extracted by the query. This is a novel take to the problem as it is restrictive than the concept of *hop*;

- the presented approaches are not confined to telephony metadata but can be easily adapted to all kinds of data representing communication between two parties such as emails, instant messages or *tweets*.

1.4 Structure of the thesis

We now describe the structure of this thesis:

- Chapter 2 provides the necessary theoretical background;
- Chapter 3 defines the considered SQL database;
- Chapter 4 contains an overview of the previous work on the subject;
- Chapter 5 presents the *graph-depth* approach;
- Chapter 6 presents the *cost-depth* approach;

- Chapter 7 describes the implementation of the system;
- Chapter 8 presents quantitative results on the system performance;
- Chapter 9 presents our conclusions.

CHAPTER 2

BACKGROUND

2.1 Relational databases

The focus of this work is relational databases. We provide the necessary terminology and notation, along with the principles of databases with *access limitations*. Several parts of this section are extracted from [8–10].

2.1.1 Preliminary definitions and notation

Intuitively, relations are sets of tuples of values belonging to given domains. The domains we consider are *abstract* in the sense that they represent concepts, like name or surname, rather than data types such as `float` or `String`. A *relation schema* is a signature of the form $r(A_1, \ldots, A_n)$, where r is a relation name, n is the *arity* of the relation schema, and each A_i is an abstract domain [1]. A *database schema* (or, simply, schema) is a set of relation schemata from different relations. A *relation* over a relation schema $r(A_1, \ldots, A_n)$ is a set of tuples $\langle c_1, \ldots, c_n \rangle$ where each c_i is a value in the abstract domain A_i. A *database instance* of schema $\mathcal{C}$ is a set of relations, one over each relation schema in $\mathcal{C}$. A *predicate* over a schema $\mathcal{C}$ is an expression of the form $r(t_1, \ldots, t_n)$, where r is the name of a relation with signature $r(A_1, \ldots, A_n)$ in $\mathcal{C}$, and each t_i is either a variable or a constant belonging to the abstract domain A_i. A *sequence* of terms $t_1, \ldots, t_n$ is denoted by $\vec{t}$ and a tuple $\langle c_1, \ldots, c_m \rangle$ is denoted by $\langle \vec{c} \rangle$; Given a sequence $\vec{t}$, its length is denoted by $|\vec{t}|$.

A *conjunctive query with negation and inequality* (NICQ) or simply *query*[2] q of arity n over a schema $\mathcal{C}$ is an expression of the form $q(\vec{X}) \leftarrow L_1, \ldots, L_m$, where $q(\vec{X})$ is the *head* of the query, $|\vec{X}| = n$ and each *atom* L_i is either a (positive) predicate of the form $r(\vec{t})$ over $\mathcal{C}$, a *negated predicate* of the form $\neg r(\vec{t})$ over $\mathcal{C}$ or an *inequality* of the form $t_i \neq t_j$. The three types of predicates above are referred to as *extensional database (edb) predicates*. A query can be compactly written as $q(\vec{X}) \leftarrow conj(\vec{X}, \vec{Y})$ where $\vec{Y}$ is made of all the variables appearing in $L_1, \ldots, L_m$ that do not appear in $\vec{X}$. Given a relation schema $r(A_1, \ldots, A_n)$, a database instance D and constants $\vec{c}$, $|\vec{c}| = n$, the atom $r(\vec{c})$ *holds* in D if $\langle \vec{c} \rangle \in r$ and the atom $\neg r(\vec{c})$ holds in D if $\langle \vec{c} \rangle \notin r$. Given constants c_i and c_j, the inequality $c_i \neq c_j$ holds if c_i is different from c_j. Given a database instance D over a schema $\mathcal{C}$ and a query $q(\vec{X}) \leftarrow conj(\vec{X}, \vec{Y})$, the *answer* to q over D is the set q^D of all the tuples $\langle \vec{c} \rangle$ of constants, where $|\vec{c}| = |\vec{X}|$, for which there is a sequence of constants $\vec{d}$ such that $|\vec{d}| = |\vec{Y}|$ and each atom in $conj(\vec{c}, \vec{d})$ holds in D. Two atoms $r_1(\vec{t_1})$ and $r_2(\vec{t_2})$ in a query q are *joined* if they

[1] We use a positional notation for relations.

[2] In the following of this document, unless specified otherwise, when referring to a *query* we will always mean a NICQ.

have at least one common variable. A query q is *safe* if all the variables in its head appear in some atom of q's body. A query q is *range-restricted* if all the variables in negated predicates or inequalities also appear in at least one positive atom. The last two properties ensure that a query always returns a finite answer. In our study we will only consider NICQs which are safe and range-restricted.

Example 2.1.1 (Conjunctive query with negation and inequality). Consider the following schema and query

$$\mathcal{C} = \{r_1(A, B), r_2(B, C)\}, \quad q(x, y) \leftarrow r_1(a_1, x), r_2(x, y)$$

where $q(x, y)$ is the head of q and $conj(x, y) = r_1(a_1, x), r_2(x, y)$ is the body of q. Assume the following database instance

$$D = \{r_1 = \{\langle a_1, b\rangle, \langle a_2, b\rangle\}, r_2 = \{\langle b, c_1\rangle, \langle b, c_2\rangle\}\}$$

The answer to q over D is $q^D = \{\langle b, c_1\rangle, \langle b, c_2\rangle\}$.

An important decision problem in database theory is the *query containment problem* which decides whether, for two given queries q_1 and q_2 over schema $\mathcal{C}$, $q_1^D \subseteq q_2^D$ holds for every database instance D. This problem is decidable for NICQs, a procedure to test containment was outlined by *Ullman* [11]. Two queries q_1 and q_2 are *equivalent*, denoted by $q_1 \equiv q_2$, iff, for every database instance D, $q_1^D \equiv q_2^D$. A query q is *minimal* if there is no query q' equivalent to q obtained by removing some atoms from the body of q. The minimality of a query can be decided by checking that, for every q' obtained by removing some atoms from q, $q \equiv q'$ does not hold.

Let $q_1(\vec{X}) \leftarrow conj_1(\vec{X}, \vec{Y})$ and $q_2(\vec{Z}) \leftarrow conj_2(\vec{Z}, \vec{U})$ be two queries such that $|\vec{X}| = |\vec{Z}|$ and $|\vec{Y}| = |\vec{U}|$. A *variable renaming* θ is a function that maps every variable of q_1 into a variable of q_2. Then, q_2 is a *variant* of q_1 iff there is a variable renaming θ such that the query ${q_2}'(\theta(\vec{Z})) \leftarrow conj_2(\theta(\vec{Z}), \theta(\vec{U}))$ is equal to q_1. For instance, consider
$q_1(x, y) \leftarrow r_1(x), r_2(x, y)$ and $q_2(z, u) \leftarrow r_1(z), r_2(z, u)$, then q_1 is a variant of q_2 because using variable renaming θ such that $\theta(x) = z$ and $\theta(y) = u$ we have

$$q_1(\theta(x), \theta(y)) \leftarrow r_1(\theta(x)), r_2(\theta(x), \theta(y)) \equiv q_1(z, u) \leftarrow r_1(z), r_2(z, u) \equiv q_2$$

If q_2 is a variant of q_1 via θ, then q_1 is a variant of q_2 via θ^{-1} and $q_1 \equiv q_2$.

We adopt the following conventions for variable names: when dealing with models of queries, we use placeholders denoted by terminal letters of the alphabet with subscripts, e.g. x_j, y_k, in place of variables. Two placeholders may be equal, e.g. $x_j = y_k$, if not stated otherwise. When dealing with concrete queries, variables are also denoted by the terminal letters of the alphabet but without subscripts.

A *union of NICQs*, (UNICQ) q of arity n over a schema $\mathcal{R}$ is a set $\{q_1, \ldots, q_k\}$ of NICQs, all having arity n. Given a database instance D, the answer q^D to a UNICQ q over D is the union of the answers over D to each query in q.

A *nr-datalog$^\neg$ program* over a database schema $\mathcal{C}$ is a sequence $q_1, \ldots, q_n$ of rules of the form $q_i(\vec{X}_i) \leftarrow body_i$. For our purposes we will only consider programs where every q_i is a *unary* NICQ. Aside from edb predicates, the body of a NICQ q_j in a nr-datalog$^\neg$ program can feature *intensional database (idb) predicates* of the form $q_i(\vec{t})$ and $\neg q_i(\vec{t})$, where $|\vec{t}|$ is the arity of q_i, but only if $i < j$. Let $Q \equiv q_1, \ldots, q_n$ be a nr-datalog$^\neg$ program over schema $\mathcal{C}$, then, for $1 \leq i \leq n$, for each database instance D over $\mathcal{C}$ and constants $\vec{c}$ where $|\vec{c}|$ is the arity of q_i, idb atom $q_i(\vec{c})$ *holds* if $\langle \vec{c} \rangle \in q_i^D$ and atom $\neg q_i(\vec{c})$ *holds* if $\langle \vec{c} \rangle \notin q_i^D$. Programs in nr-datalog$^\neg$ are equivalent to the use of *views* in SQL, where complex relations are defined by queries and then used in other queries. Let $q(\vec{X}) \leftarrow conj(\vec{X}, \vec{Y})$ be a query over $\mathcal{C}$. If the atom $q(\vec{t})$ is used in the body of a query q', $|\vec{t}| = |\vec{X}|$, we can replace $q(\vec{t})$ with $conj(\vec{t}, \vec{s})$ where $\vec{s}$ are variables which do not appear in q'. This transformation can be employed in query minimization.

2.1.2 Expressive power of NICQs

UNICQs have the same expressive power as *nonrecursive* SQL where the only comparison operators are `=` and `<>`. If we also remove the `UNION` and `OR` operators, we obtain the power of NICQs. We now show a NICQ query and a possible SQL translation.

Example 2.1.2 (Translating from NICQ to SQL). Given the database schema $\mathcal{C} = \{r(A, B), s(B, C)\}$, the query $q(x, y) \leftarrow r(a, x), s(x, y), \neg r(a, y), x \neq y$, where a denotes the string `John`, is translated into the following SQL query:

```
SELECT r.B, s.C
FROM r, s
WHERE r.A = 'John' AND r.B = s.B AND r.B <> s.C
      AND NOT EXISTS (SELECT r.A AS a, r.B AS b
                      FROM r
                      WHERE a = 'John' AND b = s.C)
```

Listing 2.1: SQL query for example 2.1.2

A thorough treatment of conjunctive queries, nr-datalog$^\neg$ and their properties can be found in *Abiteboul et al.* [12].

2.1.3 Databases with access limitations

Given a relation schema $r(A_1, \ldots, A_n)$ an *access pattern* α for q is a sequence of length n over $\{i, o\}$ such that $\alpha[k] = i$ denotes that the k^{th} argument of r is an *input* argument, an *output* argument otherwise. In this context a relation schema $r(A_1, \ldots, A_n)$, together with its access pattern α, is denoted by $r^\alpha(A_1, \ldots, A_n)$. If, for $k = 1, \ldots, n$, $\alpha[k] = o$ then r is said to be *free*. The input arguments indicated in the access pattern are those that must be bound to a value in order to query the relation. For example, in $r^{ioo}(A_1, A_2, A_3)$, the first

argument, is an input argument and a constant value in A_1 must be provided for r to be queried, while the second and third arguments are output arguments and thus constant values in A_2 and A_3 respectively do not need to be provided. The concepts of database schema, relation, database instance and query are defined as in Section 2.1.1.

An *access* is the smallest operation that can be performed on relations with access limitations, and consists of the evaluation of a query with one body atom over a relation, where all the input attributes are selected with constants, and output attributes are not selected. A relation can be accessed either if it has no input arguments or if some constants that can bind its input arguments are known. For instance an access to relation $r^{io}(A_1, A_2)$ given the constant a is the following query $\hat{r}$:

$$\hat{r}(a, x) \leftarrow r(a, x)$$

As soon as new constants are extracted with an access, these can be used to make more accesses. Such sequence of accesses constitutes an *access plan*. For each database instance, each query should then be associated with an access plan that extracts the answers to the query. This is precisely what a *query plan* does.

Example 2.1.3 (Query answering under access limitations). Consider the following schema and query:

$$\mathcal{C} = \{r_1^{io}(A, C), r_2^{io}(B, C), r_3^{io}(C, B)\}$$
$$q(x) \leftarrow r_1(a_1, y), r_2(x, y)$$

Assume that the relations are

$$r_1 = \{\langle a_1, c_1\rangle, \langle a_1, c_3\rangle\}, r_2 = \{\langle b_1, c_1\rangle, \langle b_2, c_2\rangle, \langle b_3, c_3\rangle\}, r_3 = \{\langle c_1, b_2\rangle, \langle c_2, b_1\rangle\}$$

We cannot directly answer the query because we do not know any constants to provide for the input attribute of r_2, B. The only available constant is a_1 which can be used to access r_1. The access $r_1(a_1, t)$[1] extracts the tuples $\langle a_1, c_1\rangle$ and $\langle a_1, c_3\rangle$. We have now obtained two constants, c_1 and c_3 belonging to the abstract domain C. They can be used to access r_3: from access $r_3(c_1, t)$ we extract $\langle c_1, b_2\rangle$ while from $r_3(c_3, t)$ we get nothing. Now that we have b_2 we can finally access r_2 as $r_2(b_2, t)$, obtaining the tuple $\langle b_2, c_2\rangle$. We notice that c_2 had not been obtained before, so we access r_3 once more as $r_3(c_2, t)$ which extracts the tuple $\langle c_2, b_1\rangle$. Last, access $r_2(b_1, t)$ allows us to obtain the tuple $\langle b_1, c_1\rangle$. Nothing more can be done at this point. We have then managed to extract the relations $\hat{r}_1 = \{\langle a_1, c_1\rangle, \langle a_1, c_3\rangle\}$ and $\hat{r}_2 = \{\langle b_1, c_1\rangle, \langle b_2, c_2\rangle\}$ which can be used to execute the query. The obtained answer

[1]Formally the access made to r_1 is the query $\hat{r}_1(a_1, t) \leftarrow r_1(a_1, t)$.

under access limitations is $q_\ell^D = \{\langle b_1 \rangle\}$ which is smaller than the regular answer obtained without access limitations $q^D = \{\langle b_1 \rangle, \langle b_3 \rangle\}$.

We conclude this subsection by discussing negated predicates and inequalities under access limitations. A negated predicate of the form $\neg r(\vec{t})$ cannot be dealt with in the same way as positive predicates. That is because, for $\neg r(\vec{c})$ to hold, $\langle \vec{c} \rangle$ must not belong to r. In order to verify this under access limitations, we first produce a query plan ignoring the negated atom(s) and obtain an intermediate answer then, as a further step, an access to r is made and the intermediate tuples are checked against $\neg r(\vec{t})$. Inequalities of the form $t_i \neq t_j$ are dealt with in the same way, that means a query plan ignoring them is first executed and the obtained results are checked against the inequalities, but in this case no extra accesses are required.

In general a query may have multiple query plans as some accesses may be redundant. Different query plans can be compared in terms of *cost*, which is usually defined as the number of accesses performed by the query plan. A more detailed presentation of databases with access limitations can be found in [7, 13, 14].

2.2 Graph theory

In this section we review some notions of graph theory employed in the next chapters.

Definition 2.2.1 (Directed graph)**.** A *directed graph* or, simply, *digraph* G is a pair (N, A) where $N = \{n_1, \ldots, n_{|N|}\}$ is the set of *nodes* and $A \subseteq N^2$ is the set of *arcs*. An arc a from node n_1 to node n_2 is denoted by (n_1, n_2), n_1 is referred to as the *source* of a and n_2 is referred to as the *target* of a.

Definition 2.2.2 (Adjacency)**.** Let $G = (N, A)$ be a digraph and let $n \in N$ be a node in G. The set of *adjacent nodes of* n, denoted by $Adj(n)$, is the set

$$\{m \in N \mid (n, m) \in A \vee (m, n) \in A\}$$

Definition 2.2.3 (Path)**.** Let $G = (N, A)$ be a digraph. A *path* of length k between nodes n and m is a sequence of *distinct* nodes of G $p_1, \ldots, p_{k+1}$ such that $p_1 = n$, $p_{k+1} = m$ and, for each $i = 1, \ldots, k$, $(p_i, p_{i+1}) \in A$. A *shortest path* between nodes n and m is a path of minimum length between n and m.

An efficient algorithm to find the shortest path from node i to node j is *Dijkstra's algorithm*, with running time $O(|N|^2)$.

Definition 2.2.4 (Chain)**.** Let $G = (N, A)$ be a digraph. A *chain* of length k between nodes n and m is a sequence of *distinct* nodes of G $p_1, \ldots, p_{k+1}$ such that $p_1 = n$, $p_{k+1} = m$ and, for each $i = 1, \ldots, k$, $(p_i, p_{i+1}) \in A$ or $(p_{i+1}, p_i) \in A$. A *shortest chain* between nodes n and m is a chain of minimum length between n and m.

Finding the shortest chain between nodes i and j in digraph $G = (N, A)$ is equivalent to finding the shortest *path* from i to j on digraph $G' = (N, A \cup \{(i, j) \,|\, (j, i) \in A\})$ which can be efficiently solved by means of Dijkstra's algorithm.

Definition 2.2.5 (Arc-labeled multidigraph)**.** An *arc-labeled multidigraph* is a quintuplet $G = (N, A, s, t, \ell)$ where $N = \{n_1, \ldots, n_{|N|}\}$ is the set of *nodes*, $A = \{a_1, \ldots, a_{|A|}\}$ is the set of arcs, $s : A \to N$ maps each arc to a *source* node, $t : A \to N$ maps each arc to a *target* node and $\ell : A \to \mathcal{L}$ maps each arc a to a label $\ell(a)$. Differently from digraphs, multidigraphs allow for multiple arcs to connect the same source and target. An arc $a \in A$ is denoted by (n, m, l) where $n = s(a)$, $m = t(a)$ and $l = \ell(a)$.

A sample arc-labeled multidigraph is shown in Figure 3

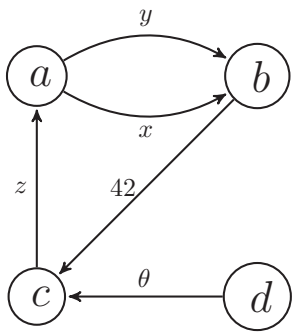


Figure 3: Sample arc-labeled multidigraph

Definition 2.2.6 (Incoming and outgoing arcs)**.** Let $G = (N, A, s, t, \ell)$ be an arc-labeled multidigraph. Let $n \in N$ be a node of G. The set of *incoming arcs* of n, denoted by $In(n)$, is the set $\{a \in A \,|\, t(a) = n\}$. The set of *outgoing arcs* of n, denoted by $Out(n)$ is the set $\{a \in A \,|\, s(a) = n\}$.

Definition 2.2.7 (Path in an arc-labeled multidigraph)**.** Let $G = (N, A, s, t, \ell)$ be an arc-labeled multidigraph. A *path of length* k from node n to node m is a sequence of *distinct* nodes of G $p_1, \ldots, p_{k+1}$ such that $p_1 = n$, $p_{k+1} = m$ and, for each $i = 1, \ldots, k$, for some arc $a \in A$, $s(a) = p_i$ and $t(a) = p_{i+1}$. A *shortest path* from n to m is a path of minimum length from n to m.

A thorough presentation of graphs can be found in *Balakrishnan* [15].

CHAPTER 3

SQL DATABASE

In this chapter we present the database schema that we assume in our analysis. We will first provide an SQL definition of the database, then we will discuss its usage with sample queries.

3.1 The *CALLS* database

We now provide a reasonable definition of an SQL database, named *CALLS* where phone numbers and phone calls between numbers are stored and identifiers can be labeled as 'RAS'[1]. Listing 3.1 shows the SQL code defining the database. At line 3 the `id` attribute of `Phone_Number` is set as *primary key*, i.e. it univocally identifies each tuple. At line 11, 15 and 16, attributes `id`, `sender` and `receiver` are set as *foreign key* to attribute `id` of `Phone_Number`, i.e. every value these attributes take must be taken by the `id` of some tuple in `Phone_Number`. For our purposes we ignore the presence of other attributes in table `Phone_Call` such as originating device, time of conversation and call duration, because they are not mentioned in the guidelines of the US administration white paper [2].

```
1  CREATE TABLE Phone_Number
2  (
3  id varchar(255) NOT NULL PRIMARY KEY, --identifier of the number
4  phone_no varchar(25), -- actual phone number
5  surname varchar(50), -- surname of owner
6  name varchar(50), -- name of the owner
7  SSN varchar(9) -- SSN of the owner
8  );
9  CREATE TABLE RAS
10 (
11 id varchar(255) FOREIGN KEY REFERENCES PhoneNumber(id)
12 );
13 CREATE TABLE Phone_Call
14 (
15 sender varchar(255) FOREIGN KEY REFERENCES PhoneNumber(id),
16 receiver varchar(255) FOREIGN KEY REFERENCES PhoneNumber(id)
17 -- additional non relevant attributes
18 );
```

Listing 3.1: Database definition

3.2 Database usage

The two query depth approaches developed in this thesis assume input queries to be NICQs. In SQL this translates into queries with no recursion and allowing only the = and <>

[1] Reasonable articulable suspicion.

comparison operators. We now provide some examples of queries reflecting NSA's usage of telephony metadata. We start with some simple queries involving tables `Phone_Call` and `RAS`.

```
SELECT R.receiver
FROM RAS AS S, Phone_Call AS R
WHERE S.id = R.sender
```

Listing 3.2: A simple query

The above query returns all the identifiers of the people who received a call from a RAS identifier.

```
SELECT R1.receiver
FROM RAS AS S1, Phone_Call AS R1, Phone_Call AS R2, RAS AS S2
WHERE S1.id = R1.sender AND R1.receiver = R2.receiver AND R2.sender = S2.id
      AND S1.id <> S2.id
```

Listing 3.3: Another simple query

The above query returns all the identifiers of the people who were called by two distinct RAS identifiers. Let us now suppose that the analyst wants to retrieve the SSN, Name and Surname of the identifiers returned by query Listing 3.2. To achieve this the following view is defined:

```
CREATE VIEW Found AS
SELECT R.receiver AS id
FROM RAS AS S, Phone_Call AS R
WHERE S.id = R.sender
```

Listing 3.4: View for query in listing 3.2

The desired information is then retrieved with the following expanded query:

```
SELECT P.id, P.SSN, P.name, P.surname
FROM Found AS F, Phone_Number AS P
WHERE F.id = P.id
```

Listing 3.5: Expanded query

We now consider that the intelligence analyst may want to look into the phone calls of a small *seed* of RAS identifiers. This can be achieved by defining the following VIEW on table `RAS`, where `id_1`,`id_2`,...,`id_n` are some known identifiers in `RAS`:

```
CREATE VIEW Seed AS
SELECT id
FROM RAS
WHERE id = id_1
   OR id = id_2
   ...
   OR id = id_n
```

Listing 3.6: Definition of a seed

Suppose the analyst wants to retrieve all the identifiers of people who were called by a member of the above `Seed`. The following query achieves this goal:

```
SELECT R.receiver
FROM Seed AS S, RAS AS R
WHERE S.id = R.sender
```

Listing 3.7: Query with seed

CHAPTER 4

STATE OF THE ART

In this chapter we will provide an overview of the input driven approach proposed in *Kanich et al.* [3] and the work in *Panebianco* [4].

4.1 Input driven approach

In 2013, *Kanich et al.* have proposed an *input driven* approach to ensure the legality of queries on telephony metadata [3]. Identifiers are considered legally queryable according to the guidelines in the white paper [2], where it is stated that the NSA is allowed to query identifiers up to three *hops* from the RAS identifiers. The following graph-theoretic definition is adopted: "let the nodes represent phone numbers, and edges represent phone calls in the BR metadata. For two nodes to be three "hops" apart, all queries must be performed on the subgraph represented by the union of all calls among nodes for which a path of length three or less exists between a RAS identifier and a candidate node" [3, p. 3]. The approach generalizes this requirement considering the maximum number of allowed hops to be a generic integer $\bar{k} > 0$. The approach is input driven because it involves the computation of the maximal set of *legally queryable identifiers* (LQI). This set is comprised by all and only those identifiers that lie at $k \leq \bar{k}$ hops from RAS identifiers. A query is guaranteed to comply with the above order in the following way: before any query is executed, the `Phone_Call` table is filtered by removing all the records where the sender or the receiver identifier does not appear in LQI. This way, the query can only retrieve identifiers which lie at a legal number of hops from the RAS identifiers.

We now formalize the above. Consider database schema $\mathcal{C} = \{R(s, r), S(id)\}$ where R corresponds to table `Phone_Call` in Chapter 3, S represents table `RAS`, s and r represent the `sender` and `receiver` attributes of `Phone_Call` respectively and id represents the `id` attribute of `RAS`.

Definition 4.1.1 (Maximal set of legally queryable identifiers)**.** Let D be a database instance over database schema $\mathcal{C}$ and let $\bar{k}$ be the maximum number of hops allowed by the law. The *maximal set of legally queryable identifiers*, denoted by $\hat{I}_{\bar{k}}^{D}$, is the set $\bigcup_{k=0}^{\bar{k}} I_k^D$ where all I_k are query unions of arity 1 such that, for every database instance D,

- $I_0^D = \{q_0^D\}$ where q_0 is the query $q_0(x) \leftarrow S(x)$;
- $I_k^D = \{q_{k,s}^D, q_{k,r}^D\}$ where
  - $q_{k,s}$ is the query $q_{k,s}(x) \leftarrow R(x, y), I_{k-1}(y)$;
  - $q_{k,r}$ is the query $q_{k,r}(x) \leftarrow R(y, x), I_{k-1}(y)$.

for $1 \leq k \leq \bar{k}$.

Intuitively, $\hat{I}^D_{\bar{k}}$ is the set made of all the identifiers which lie at $k \leq \bar{k}$ hops from RAS. After $\hat{I}^D_{\bar{k}}$ is computed, each database instance D over $\mathcal{C}$ is filtered in the following way:

Definition 4.1.2 (Filtered database instance)**.** Let $D = \{S, R\}$ be a database instance over $\mathcal{C}$ and let $\bar{k}$ denote the maximum number of hops allowed by the law. The *filtered* D, denoted by $D^{\bar{k}}$, is the database instance $\{S^{\bar{k}}, R^{\bar{k}}\}$ such that

- $S^{\bar{k}} = S$;
- $R^{\bar{k}} = \{\langle i, j\rangle \in R \,|\, \{\langle i\rangle, \langle j\rangle\} \subseteq \hat{I}^D_{\bar{k}}\}$.

The filtered relation $R^{\bar{k}}$ contains all and only the phone calls that can be queried in compliance with the law. The approach simply imposes that, for every query q and database instance D, $q^{D^{\bar{k}}}$ is computed instead of q^D.

4.1.1 Pros and cons of the approach

This approach guarantees that, whichever query is executed, the analyst will only be able to see legally queryable identifiers. Consequently, the analyst can design input queries without worrying about their compliance to the requirements. The drawback of this approach is that, for each database instance D, it requires the computation of $\hat{I}^D_{\bar{k}}$ which requires multiple joins involving table Phone_Call. Since billions of phone calls are made every day in the US, we believe it is reasonable to assume the presence of millions of records in Phone_Call [16]. This fact makes the computation of $\hat{I}^D_{\bar{k}}$ a quite expensive operation which has to be performed for each database instance D considered by the analyst. Moreover, after $\hat{I}^D_{\bar{k}}$ is computed, the filtered database instance $D^{\bar{k}}$ must be computed as well. The cost of this computation clearly depends on the size of D. To sum up the above, the whole approach becomes more and more expensive as the size of Phone_Call grows.

4.2 Experiments on graph databases

The work in *Panebianco* [4] aimed at finding technologies alternative to SQL relational databases, to store and query telephony metadata. The author proposed *graph databases* as a solution that proved to have faster execution of queries than SQL databases and also provided a more natural way to represent phone calls. In graph databases, information is stored in the form of a graph and in the proposed database nodes represented identifiers and edges represented phone calls. Because of the ease in the representation of the domain and the good performance obtained when executing queries retrieving all identifiers at k hops from RAS, this study contradicted the NSA's claim that enforcing legal access was technically impossible [1].

CHAPTER 5

GRAPH-DEPTH APPROACH

In this chapter we present the *graph-depth* approach to the problem of ensuring the legality of queries on database *CALLS*. In Section 5.1 we define the formal database schema and the admissible queries, which comply with the following order from [2]:

"Under the FISC orders authorizing the collection, authorized queries may only begin with an "identifier", such as a telephone number, that is associated with one of the foreign terrorist organizations that was previously identified to and approved by the Court. An identifier used to commence a query of the data is referred to as a "seed"." [2, p. 3].

In Section 5.2 we define the notion of graph-depth of a query, which represents the maximum number of hops at which any identifier returned by the query lies from the RAS identifiers. The graph-depth approach employs this notion to verify the compliance of an admissible query with respect to the following order from [2]:

"Information responsive to an authorized query could include, among other things, telephone numbers that have been in contact with the terrorist-associated number used to query the data, [...]. Under the FISC's order, the NSA may also obtain information concerning second and third-tier contacts of the identifier (also referred to as "hops"). The first "hop" refers to the set of numbers directly in contact with the seed identifier. The second "hop" refers to the set of numbers found to be in direct contact with the first "hop" numbers, and the third "hop" refers to the set of numbers found to be in direct contact with the second "hop" numbers. Following the trail in this fashion allows focused inquiries on numbers of interest, thus potentially revealing a contact at the second or third "hop" from the seed telephone number that connects to a different terrorist-associated telephone number already known to the analyst. Thus, the order allows the NSA to retrieve information as many as three "hops" from the initial identifier." [2, p. 3-4].

Our approach allows to verify more general requirements than the above, specifically that a query expands no more than k hops, for any $k \geq 0$. In Section 5.4 we propose an algorithm to compute the graph-depth of an admissible query and in Section 5.5 we describe the overall approach to the problem. In Section 5.6 we compare this approach with the state of the art. Finally, in Section 5.7 we discuss our approach in the context of recursive queries.

5.1 Framework definition

We now define the database schema and the queries that we will base our study on.

The physical database proposed in Chapter 3 can be mapped into the following database schema:

$$\mathcal{C} = \{S(id), R(s, r)\}$$

Here S represents table RAS, R represents table Phone_Call and id, s and r represent the same abstract domain (identifiers)[1]. We also assume the use of a set $\mathbb{E}$ of *seeds* S_j:

$$\mathbb{E} = \{S_1(id), \ldots, S_n(id)\},\ S_j \subset S,\ j = 1, \ldots, n.$$

These may be used in case the searches are started from a restricted list of suspects. We also assume that the RAS table S and any seed S_j are never used in the same query. We also let S_0 be S. We use subscripts both to identify seeds and to identify the various uses of R in queries e.g. $S_2(x)$ is a predicate over relation S_2 while $R_3(y, z)$ is the third use of a predicate over R.

The queries we will consider are NICQs over $\mathcal{C}$ which are safe and range-restricted. Nr-datalog$^\neg$ programs will be made of unary NICQs over $\mathcal{C}$ with the addition of *idb* predicates over such queries. We will denote the j-th query of a nr-datalog$^\neg$ program as q_j.

To a query q over $\mathcal{C}$ we can associate a digraph, named the *call graph* of q, which encodes the way in which identifiers from the same tuple of q^D are related via phone calls.

Definition 5.1.1 (Call graph)**.** Let $q(\vec{X}) \leftarrow conj(\vec{X}, \vec{Y})$ be a query over $\mathcal{C}$. The *call graph* of q is the digraph $G_q = (N_q, A_q)$ where N_q consists of the set of variables appearing in q and $(x_j, y_j) \in A_q$ iff atom $R(x_j, y_j)$ is in $conj(\vec{X}, \vec{Y})$. The node set N can be partitioned into the following sets:

- $B_q = \{x_i \,|\, \text{for some } j \text{ atom } S_j(x_i) \text{ appears in } q\text{'s body}\}$
- $K_q = \{x_i \,|\, x_i \notin B_q \text{ and for some } j \text{ atom } q_j(x_i) \text{ appears in } q\text{'s body}\}$
- $I_q = N_q \setminus (B_q \cup K_q)$

where B_q represents the *Bad*, K_q represent those *Known* through previous queries and I_q represents the *Innocent*.

Intuitively an arc from node x_i to node x_j represents a call between *[the person identified by]* x_i and *[the person identified by]* x_j. Negated atoms and inequalities are not encoded in the call graph as they do not relate variables via phone calls: the former relate identifiers which did not interact and the latter relate distinct identifiers.

Example 5.1.1 (Call graph of a query). For the following query

$$\begin{aligned} q(x, y, z, u, v) \leftarrow & S_0(x), S_1(z), R_1(x, y), R_2(y, z), \neg R_3(x, z), \\ & R_4(y, u), S_2(u), R_5(v, u), R_6(v, w), u \neq w \end{aligned}$$

where $\mathbb{E} = \{S_1(id), S_2(id)\}$, the call graph of q is shown in Figure 4. In G_q, $N_q = \{x, y, z, u, v\}$, $A_q = \{(x, y), (y, z), (y, u), (v, u), (v, w)\}$, $B_q = \{x, z, u\}$ and $I_q = \{y, v, w\}$.

---

[1] Table Phone_Number does not contain information relevant to the problem and is therefore ignored.

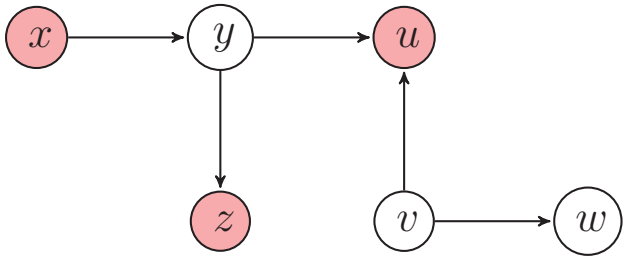


Figure 4: Call graph for example 5.1.1 (nodes in B are red).

We can now express the order [2, p. 3] in our framework by defining the notion of *admissible query*.

Definition 5.1.2 (Admissible query)**.** A query $q(\vec{X}) \leftarrow conj(\vec{X}, \vec{Y})$ over $\mathcal{C}$ is *admissible* if and only if the following all hold

1. every connected component of the call graph G_q contains a node in $B_q \cup K_q$;
2. no constants appear in q.

The first condition ensures that every node in the call graph is reachable from a bad node. This is trivial if the node is reachable from some bad node, but it also is the case if it is reachable from a known node, provided that every query in the program is admissible. The second condition comes from the observation that, based on [2, p. 3], constant values must identify RAS identifiers. This allows a constant c to be removed from an admissible query q in the following way:

1. we define a new seed $S_c(id)$ such that $S_c \equiv \{\langle c \rangle\}$;
2. we add the atom $S_c(x_c)$ to the query, where x_c is a new variable;
3. we replace all the occurrences of c in q with x_c.

The above procedure is equivalent to *deskolemization* in first order logic and preserves satisfiability. Removing constants simplifies our exposition.

We now show with an example that admissibility is necessary to respect the order.

Example 5.1.2 (Non admissible query)*.* Consider the following CQ, where $\mathbb{E} = \emptyset$:

$$q(x, y, z, u) \leftarrow S(x), R_1(x, y), R_2(z, u)$$

If executed, this query will return tuples $\langle c_1, c_2, c_3, c_4 \rangle$ of constants such that $\langle c_1 \rangle \in S$, $\langle c_1, c_2 \rangle \in R$ and $\langle c_3, c_4 \rangle \in R$. Identifier c_1 refers to a suspect, c_2 is related to c_1 via a

phone call but neither c_3 nor c_4 are explicitly related to suspects via other phone calls. This clearly violates the order in [2, p. 3] and is reflected in the call graph G_q, in Figure 5, where $B = \{x\}$, $I = \{y, z, u\}$ and there is no chain between x and z or between x and u.

Figure 5: Call graph for example 5.1.2.

5.2 Definition of graph-depth

In this section we provide a definition of *graph-depth* of an admissible query. The *graph-depth* of a query is the maximum number of hops, as defined in [2, p. 3], that the query traverses in the *CALLS* database upon execution. In Chapter 4 we have seen that the set of telephony metadata records can be interpreted as a graph where nodes are identifiers and arcs are phone calls between identifiers. We have also seen earlier in this chapter that we can represent the answer to a query q via its call graph G_q. Therefore, to say that an identifier i lies at k hops from S is equivalent to say that i is at distance $d(i) = k$ from S [1]. If an identifier i is not connected to S then $d(i) = \infty$.

Definition 5.2.1 (Graph-depth)**.** Let $q(\vec{X}) \leftarrow conj(\vec{X}, \vec{Y})$ be an admissible query over $\mathcal{C}$ and $\mathcal{D}$ be the set of all database instances of $\mathcal{C}$. The *graph-depth* of q, denoted by $\delta(q)$ is defined as

$$\delta(q) = \max_{D \in \mathcal{D},\, \langle \vec{c} \rangle \in q_f^D,\, i \text{ in } \vec{c}} d(i) \tag{5.1}$$

where q_f is the query $q_f(\vec{X}, \vec{Y}) \leftarrow conj(\vec{X}, \vec{Y})$ having the same body as q and all the variables of q in its head.

The above definition states that the graph-depth of q is the maximum distance of an identifier returned by the q_f over all database instances D and over all output tuples of the

[1] For distance between i and S we mean the length of the shortest chain between i and any identifier $b \in S$.

answer q_f^D, where query q_f is the same as q but where all the variables are in the head. Employing q_f in the definition is needed to make the graph-depth independent from the head of the query. As an example, queries $q_1(y) \leftarrow S(x), R(x,y), R(y,z)$ and $q_2(x) \leftarrow S(x), R(x,y), R(y,z)$ have the same graph-depth of 2 even if, e.g. in every tuple $\langle c \rangle \in q_1^D$, $d(c) = 1$ because, in both cases, to obtain such an answer, distance-2 identifiers must be found (variable z). The graph-depth $\delta(q)$ represents a *pessimistic* estimate of the number of hops traversed by the query. For instance every identifier i returned by the query q_1 above lies at most 1 hop from the seed because it is the receiver of a call originated by a RAS identifier, but i could be a RAS identifier itself thus lying 0 hops from RAS. It is also clear that $\delta(q)$ is always a finite number because, in admissible queries, all the retrieved identifiers are connected to the seed.

In the case of *union of queries* the definition of graph-depth can be trivially extended:

Definition 5.2.2 (Graph-depth of union)**.** Let $q = \{q_1, \dots q_n\}$ be a union of admissible queries. Then the *graph-depth of* q, denoted by $\delta(q)$, is defined as

$$\delta(q) = \max_{i=1,\dots,n} \delta(q_i) \tag{5.2}$$

The rationale behind this definition is simple: let $q = \{q_1, \dots q_n\}$ be a union of admissible queries and i an identifier contained in q^D, where D is a database instance of $\mathcal{C}$. The number of hops required to retrieve i is inside the set $\{\delta(q_1), \dots, \delta(q_n)\}$ so, in order to provide a conservative assessment of the graph-depth of q, the maximum of those values is taken.

We conclude this section by discussing how the graph-depth of an admissible query can be computed when the query belongs to a nr-datalog$^\neg$ program. Let $Q \equiv q_1, \dots, q_n$ be a nr-datalog$^\neg$ program over $\mathcal{C}$, where all the q_j are admissible queries. Inside a nr-datalog$^\neg$ program, every q_j, $j = 2, \dots, n$, allows for *idb* predicates of the form $q_i(\vec{t})$ and $\neg q_i(\vec{t})$, $i < j$ to appear in its body. Positive idb atoms could be replaced with a sequence of edb atoms: given the query $q_i(\vec{X}) \leftarrow conj_i(\vec{X}, \vec{Y})$, atom $q_i(\vec{t})$ can be replaced by $conj_i(\vec{t}, \vec{s})$, where $\vec{s}$ are new variables. If $conj_i$ also contains positive idb atoms, this substitution can be recursively applied until no positive idb atoms appear in the body of q_j. Although this would bring us to the base problem of evaluating the graph-depth of a single query, we will not employ the above transformation but will instead treat every positive idb atom as a seed of id's whose distance from S has been already established. Negated idb atoms will be trated in the same way. In both cases idb predicates influence the graph-depth of the current query q_j: in order to answer a query q_j where predicate $(\neg)q_i(\vec{t})$ is used, query q_i must be answered to beforehand and thus its graph-depth $\delta(q_i)$ does impact on $\delta(q_j)$. This means that $\delta(q_i)$ must be computed by counting the number of hops expanded by q_j considering the edb atoms in its body *and* comparing this number with $\delta(q_i)$ for every i s.t. $(\neg)q_i(\vec{t})$ appears in

the body of q_j. The graph-depth of q_j is the maximum of these values. We now show an example to clarify the above.

Example 5.2.1 (Graph-depth and nr-datalog$^\neg$ programs). Consider the following nr-datalog$^\neg$ program Q:

$$q_1(y) \leftarrow S(x), R(x, y)$$
$$q_2(y) \leftarrow q_1(x), R(x, y), \neg q_1(y)$$

Clearly $\delta(q_1) = 1$. Query q_2 features predicate q_1 both in its positive and negated form. This implies that $\delta(q_2) \geq \delta(q_1)$. Since q_1 outputs 1-hop id's, the path "x calls y" in q_2 expands 2 hops. The graph-depth of q_2 is therefore the maximum value between $\delta(q_1)$ and the hops expanded by q_2 itself hence $\delta(q_2) = \max(1, 2) = 2$.

5.3 Query minimization and graph-depth computation

In this section we will discuss how query minimality can affect the graph-depth of the query. We will show an example of a query whose graph-depth is reduced after minimization.

Example 5.3.1 (Query minimization and graph-depth). Consider the following query:

$$q(z) \leftarrow S(x), R(x, y), R(y, z), S(z), R(y, t)$$

Its graph-depth $\delta(q)$ is 2 since t represents an identifier called by y which is in turn called by a RAS identifier x. By applying the minimization algorithms described in [11, 12] it turns out that $R(y, t)$ is redundant and can be removed without altering the answer to the query. The minimal query equivalent to q is therefore

$$q_m(z) \leftarrow S(x), R(x, y), R(y, z), S(z)$$

The graph-depth $\delta(q_m)$ is 1 as x and z are RAS identifiers (distance 0) and y called and was called by both (distance 1).

The example above shows that in order to obtain the real graph-depth of a given query, the query must be minimized beforehand. In the case of nr-datalog$^\neg$ programs, further minimization could be possible if idb predicates were transformed into sequences of edb atoms as hinted earlier. We will not perform such transformation to keep the query preprocessing phase relatively short and because a competent analyst can look into the problem directly by rewriting non minimal queries. This choice implies that idb atoms will be trated as edb atoms during the minimization phase.

5.4 Proposed algorithm

We are now ready to provide an algorithm to compute the graph-depth of an admissible query q_j. The general idea has been already outlined informally in the previous sections:

to compute the distance between innocent and bad nodes in the call graph G_q we find the innocent node whose shortest chain to S is the longest. This computation can be carried over efficiently by adapting Dijkstra's famous algorithm. We also have to take care of minimizing the query before this computation and of handling idb predicates, in case the query is part of a larger program. We now present the *ComputeGraphDepth* algorithm (algorithm 1) which computes the graph-depth of an admissible query. At line 1 we define

Algorithm 1: *ComputeGraphDepth*

Input: An admissible query q_j over $\mathcal{C}$
Output: a non negative integer Δ representing $\delta(q_j)$

1 $\Delta \leftarrow 0$
2 $q_j \leftarrow$ the minimal query equivalent to q_j
3 **for** *every idb atom* $(\neg)q_i(x_i)$ *in* q_j*'s body* **do**
4 $\quad \Delta \leftarrow \max(\Delta, \delta(q_i))$
5 **end**
6 $G_{q_j}(N_{q_j}, A_{q_j})$: the call graph of q_j
7 D_j: array $[1, \ldots, |N_{q_j}|]$ of $\mathbb{N}$ **init** $\begin{cases} 0 & v \in B_{q_j} \\ D_i[q_i\text{'s head variable}] & v \notin B_{q_j} \text{ and } q_i(v) \text{ in } q_j\text{'s body} \\ \infty & \text{otherwise} \end{cases}$
8 $visited$: array $[1, \ldots, |N_{q_j}|]$ of booleans **init** $\begin{cases} \textbf{true} & i \in B_{q_j} \\ \textbf{false} & \text{otherwise} \end{cases}$
9 U: queue of nodes **init** all B_{q_j} nodes
10 **while** $\neg$ `empty(`U`)` **do**
11 $\quad v \leftarrow$ `dequeue(`U`)`
12 $\quad$ **for** u *in* v*'s adjacency list* **do**
13 $\quad\quad$ **if** $\neg visited[u]$ **then**
14 $\quad\quad\quad visited[u] \leftarrow$ **true**
15 $\quad\quad\quad D_j[u] \leftarrow \min(D_j[v] + 1, D_j[u])$
16 $\quad\quad\quad \Delta \leftarrow \max(\Delta, D_j[u])$
17 $\quad\quad\quad$ `enqueue` (U, u)
18 $\quad$ **end**
19 **end**
20 **return** Δ

the variable Δ, which holds the graph-depth computed so far. At line 2, q_j is minimized is performed to avoid graph-depth overestimation. Lines 3-5 update the value of Δ by taking into account all idb atoms. Here we suppose that, if the query is part of a nr-datalog$^\neg$ program, $\delta(q_i)$ has been already computed for all the previous queries q_i, $i < j$. At line 6 we build the call graph of the minimized query q_j, while at lines 7-9 three useful data structures are defined. D is an array keeping track of the distance between all the nodes in N_{q_j}. If a node v is in B_{q_j} then its distance from S is of course 0 while, if $v \notin B_{q_j}$ and v appear in a positive idb atom q_i, its current distance is actually the distance of the head variable of q_i, obtained while computing $\delta(q_i)$ and stored in D_i. In every other case, $D_j[v]$ is set to ∞. The array $visited$ stores, for each node, whether it has been already visited or not[1]. Finally, U is a priority queue used to keep track the next nodes whose neighbors need to be visited. Lines 18-27 contain the core of the algorithm: we dequeue the current node and we update the distance of the not yet visited neighbors, then we enqueue them and update the graph-depth so far (Δ) if the new value is greater then previously encountered. At the end of the computation, $\Delta = \delta(q)$ is returned.

Time complexity of *ComputeGraphDepth*

We now assess the time complexity of the algorithm. At line 2, the input query is minimized resulting in the minimal query q_j. The minimization problem for NICQs is complete for the class Π_2^p [11] which is coNP with NP oracle[2]. The **for** loop at lines 3-5 performs one scan of the body of q_j and therefore takes at most p iterations, where p is the number of atoms in q_j's body. At line 6, the call graph of q_j is built. This operation requires one scan of the query's body which has, again, p atoms. Initializing D and $visited$ takes a time $\Theta(|N_{q_j}|)$ and initializing U takes a time $\Theta(|B_{q_j}|)$. Overall, lines 7-9 have a complexity of $\Theta(|N_{q_j}|)$. We now analyze lines 10-19, where the core of the algorithm resides: since a node cannot be inserted into the queue more than once and all the nodes in G_{q_j} are chained, the **while** loop will run for $|N_{q_j}|$ iterations. The **for** loop scans all the adjacent nodes to v. Whenever a new node is found, it is marked as visited and enqueued. Based on the above observations, the **while** loop has a complexity of $\Theta(|A_{q_j}|)$. Overall, the time complexity of *ComputeGraphDepth* is the sum of the miminimization time complexity (Π_2^p-complete), query body scan and data structure initialization ($\Theta(p + |N_{q_j}|)$) and the "Dijkstra" part ($\Theta(|A_{q_j}|)$). The sum $\Theta(p + |N_{q_j}| + |A_{q_j}|) = \Theta(p + |N_{q_j}|)$[3] is negligible with respect to the much more expensive minimization, which determines the overall complexity of *ComputeGraphDepth*. We point out that the skilled analyst may not require query minimization, which would make

[1] All B_{q_j} nodes are recorded as being already visited because their distance from B_{q_j} is trivially 0

[2] Problems in Π_2^p can be expressed as $\{w \mid \forall x \exists y \, \phi(w, x, y)\}$, where x and y are strings of length bounded by a polynomial of the length of w and ϕ is a function that can be computed in polynomial time.

[3] The number of arcs in G_{q_j} is equal to the number of R-atoms in q_j.

the overall procedure linear in the size of the query ($\Theta(p + |N_{q_j}|)$). We will show some quantitative results on the algorithm performance in Chapter 8.

5.5 Description of the approach

The graph-depth approach to ensure the legality of queries on telephony metadata is fairly simple: suppose that the current laws allow queries to expand no more than $\bar{k}$ hops from the RAS identifiers. The graph-depth approach works as follows:

1. Given an admissible query $q(\vec{X}) \leftarrow conj(\vec{X}, \vec{Y})$ over $\mathcal{C}$, *ComputeGraphDepth* is run on q, returning $\delta(q)$;
2. If $\delta(q) \leq \bar{k}$, the query complies with the above requirement and can be executed;
3. If $\delta(q) > \bar{k}$, the query *may* violate the above requirement and, in order to execute it, some corrective action must be performed, such as applying the input driven approach described in *Kanich et al.* [3].

5.6 Comparison with the input driven approach

In this last section we compare the graph-depth approach with the approach proposed in *Kanich et al.* [3]. We first show that the two approaches are correlated, then we discuss the reasons why each approach may be preferred over the other.

5.6.1 Relation between the two approaches

We present a theorem stating the link between the input driven approach and the graph-depth approach.

Theorem 5.6.1 (Input driven approach and graph-depth approach)**.** *Let q be an admissible query over $\mathcal{C}$. Let D be a database instance over $\mathcal{C}$. Let $D^{\bar{k}}$ be the* filtered database instance *with maximum number of hops k as described in Section 4.1. If, for some $k \geq 0$, $q^D \not\equiv q^{D^{\bar{k}-1}}$ and $q^D \equiv q^{D^{\bar{k}}}$, then $\delta(q) \geq k$.*

Proof. If $q^D \not\equiv q^{D^{\bar{k}-1}}$ and $q^D \equiv q^{D^{\bar{k}}}$ we have that, for every c_i in $\langle \vec{c} \rangle \in q^D$, $d(c_i) \leq k$ and for some c_i in $\langle \vec{c} \rangle \in q^D$, $d(c_i) = k$. This means that

$$\max_{\langle \vec{c} \rangle \in q^D,\, c_i \text{ in } \vec{c}} d(c_i) = k \tag{5.3}$$

therefore when taking the maximum of (Equation 5.3) over all possible database instances we have

$$\max_{D \in \mathcal{D},\, \langle \vec{c} \rangle \in q^D,\, c_i \text{ in } \vec{c}} d(c_i) = \max\{\ldots, k, \ldots\} \geq k \tag{5.4}$$

but the LHS of (Equation 5.4) is exactly the definition of $\delta(q)$ in (Equation 5.1) hence $\delta(q) \geq k$. □

In general, if $q^D \equiv q^{D^{\bar{k}}}$, nothing can be said about $\delta(q)$. We show this with an example.

Example 5.6.1 (Input driven approach and graph-depth approach). Consider the following query

$$q(x, y, z) \leftarrow S_0(x), R_1(x, y), R_2(y, z)$$

and suppose the law allows $\bar{k} \leq 1$. Let D be the database instance where $S = \{\langle b \rangle\}$ and $R = \{\langle b, i_1 \rangle, \langle b, i_2 \rangle, \langle i_1, i_2 \rangle\}$. It turns out that $D^1 = \{S^1, R^1\}$ where $S^1 \equiv S$ and $R^1 \equiv R$, therefore $q^D \equiv q^{D^1} \equiv \{\langle b, i_1, i_2 \rangle\}$. We may think that, because of this result, $\delta(q) \leq 1$. However, if we consider a different database instance D' such that $S = \{\langle b \rangle\}$ and $R = \{\langle b, i_1 \rangle, \langle i_1, i_2 \rangle\}$, we have that $R^1 \equiv \{\langle b, i_1 \rangle\}$, therefore $q^{D'^1} = \emptyset$. This shows that, basing solely on the fact that, for some D and k, $q^D \equiv q^{D^k}$, we cannot say anything about $\delta(q)$.

5.6.2 Advantages and disadvantages of the two approaches

To conclude, we compare the strengths and weaknesses of the two approaches. The graph-depth approach allows a precise measurement of the compliance of a query w.r.t. the order in [2], which is independent from the specific database instance the query is executed on. Instead, the input driven approach can only provide a lower bound to the graph depth of a query and, while it filters each database instance to ensure compliance, it actually allows the execution of queries which are not always compliant to the above guidelines: example 5.6.1 shows that different databases can lead to different evaluations of the compliance of a query. On the practical side, if the analyst is interested in obtaining all the results he/she can get no matter the query, the graph-depth approach does not provide this feature while the input driven approach is specifically devoted to it. Conversely, if the intelligence agency is interested in automatizing the analysis on daily telephony data by defining a list of standard queries to execute on every database of interest, the graph-depth approach provides knowledge on the worst-case-scenario behavior of every query. This gives the agency the ability to craft a set of *safe* queries and also frees them from the burden of filtering any database instance.

5.7 Adding recursion

As a last remark, we discuss the consequences of considering recursive queries in the graph-depth approach. Namely a query $q(\vec{X}) \leftarrow conj(\vec{X}, \vec{Y})$ is *recursive* if predicate $q(\vec{t})$ appears in $conj(\vec{X}, \vec{Y})$. Recursive queries are defined via 2 or more NICQs with the same head and arity. We now provide an example of recursive query and discuss its graph-depth.

Example 5.7.1 (Recursive query over $\mathcal{C}$). Consider the following recursive query:

$$q(x) \leftarrow S(x)$$
$$q(x) \leftarrow q(y), R(x, y)$$

The answer to q is the set of all the id's in S and all the id's who called some id already in q. It is clear that the graph-depth of q is infinite, because for every database instance

D we can always find some D' s.t. $\max_{i \text{ in } \langle c \rangle \in q^D} d(i) < \max_{i \text{ in } \langle c \rangle \in q^{D'}} d(i)$, namely by adding a call from a new id to the most distant id in D. It follows that the graph-depth approach cannot be effectively applied to recursive queries (or programs), which is why we will never deal with them in this work.

CHAPTER 6

COST-DEPTH APPROACH

In this chapter we present the notion of *cost-depth* as an alternative way to assess the *depth* of admissible queries over the target database *CALLS*. Although similar to the concept of graph-depth presented in the previous chapter, cost-depth captures different traits of queries as, instead of hops, we will deal in terms of *joins*. In Section 6.1 we will provide motivation and an informal overview of the subject. In Section 6.2 the formalism required to define the problem is introduced, in Section 6.3 a solution to the problem is analyzed and in Section 6.4 two algorithms to compute the cost-depth of a query are presented. Finally, in Section 6.5 the two notions of graph-depth and cost-depth are compared.

6.1 Motivation and informal exposition

The graph-depth approach, described in Chapter 5, performs a conservative assessment of the number of hops expanded by an admissible query. This is perfectly in line with the guidelines published in [2]. However, the notion of hop does not allow to distinguish between different joins which depend on the direction of a call: namely, to look at those who *called* a certain seed of RAS id's accounts for one hop just as the action of looking at those who *were called* by a certain seed of RAS id's. This implies that, if the two actions are combined in a query, the resulting graph-depth will still be one (example 6.1.1), while clearly we are using two different call patterns: those who received a call from S (y) and those who called someone in S (z).

Example 6.1.1 (1-hop query).

$$q(y, u) \leftarrow S(x), R_1(x, y), R_2(z, x), y \neq z$$

In this chapter however, we will adopt a *cost* based notion of query depth, by analyzing the different joins that are used in a query. We restrict the way in which the query engine executes a query by enforcing *access limitation* on the relation schema $R(s, r)$: in order to extract tuples from R the engine must either provide a set of values for the sender or for the receiver attribute. This means that R is not accessible as a whole (all phone calls) but only fragments can be extracted (those who called/where called by a certain known list of id's)[1]. Given a query, the goal of the cost-depth approach is to find the minimum number of *accesses* to R needed to answer the query, given the access limitations to R above. An *access* is essentially the action of joining a set of known identifiers to either side of the

[1]This is analogous to how data is presented on the web via input fields and forms: instead of showing all the `<book, author>` pairs, web sites usually require the user to provide a value for one field to extract the other information.

R relation (sender or receiver) obtaining a new set of identifiers as a result. In order to answer a query, the engine must find a way to extract the required tuples of R respecting the access limitations: a *search plan* is the assignment of an access to every occurrence of the R predicate in the query and its *cost* is evaluated according to the following:

- every access has a unitary cost;
- if an access occurs multiple times in a query, its cost is accounted for only once[1];
- the cost-depth of a query is the sum of the cost of all the (distinct) accesses needed to answer the query.

The minimum cost over all the possible search plans is the cost-depth of the query. The search for the cost-depth can be then formulated as "if the query engine had to respect the access limitations to R, what would be the minimum number of accesses necessary to answer the query?". For instance, in example 6.1.1, the known set of id's is S, which is joined to both R_1 and R_2, on the s and r attributes respectively, which translates into two distinct accesses: "obtain y as the callee of x" and "obtain z as the caller of x". Since the two patterns represent two different joins, the cost-depth of the query is 2. To compute the cost-depth of a query one must find the minimum number of accesses that answer the query, given that the initially known identifiers are all and only the id's in S.

Example 6.1.2 (Cost-depth computation).

$$q(x, y, z) \leftarrow S(x), R_1(x, y), R_2(z, y), S(z)$$

The known set of identifiers is S, we observe that the joins are S and R_1 on x, R_1 and R_2 on y and R_2 and S on z. There are three possible search plans for q:

1. We extract the $\langle x, y\rangle$ tuples by joining the known set S to R_1 and then joining R_1 and R_2, using the new set of $\langle y\rangle$ id's, to extract the $\langle z, y\rangle$ tuples. By joining the obtained $\langle x, y, z\rangle$ tuples with S on z we obtain the answer;
2. we extract the $\langle x, y\rangle$ tuples by joining the known set S to R_1 and then extract the $\langle z, y\rangle$ tuples by joining the known set S to R_2. By joining the two obtained fragments $\langle x, y\rangle$ and $\langle z, y\rangle$ on y we obtain the answer;
3. we extract the $\langle z, y\rangle$ tuples by joining the known set S to R_2 and then joining R_2 and R_1, using the new set of $\langle y\rangle$ id's, to extract the $\langle x, y\rangle$ tuples. By joining the obtained $\langle x, y, z\rangle$ tuples with S on x we obtain the answer.

[1]The reason for this is that, once an access is performed, the resulting tuples can be cached for later use.

Search plans 1 and 3 involve two distinct access patterns, while search plan 2 involves two equivalent accesses: $S(x) \to R(x, y)$ and $S(z) \to R(z, y)$[1] represent the same join between the S and R relations. The cost-depth of q is therefore 1.

The key characteristic of the cost-depth approach is that it distinguishes between the different types of joins/accesses in a query whereas the graph-depth approach only counts the number of hops, thus providing a generally smaller depth as a result.

6.2 Problem definition

The considered database schema is $\mathcal{C} = \{S(id), R(s, r)\}$, once again we allow the use of a variable number n of seeds $S_j(id)$, $S_j \subset S$, $j = 1, \ldots, n$ and let S_0 be S just like in Chapter 5. We only consider *admissible queries* as defined in the same chapter. We now define the notions of *access*, *search plan* and *plan cost*.

Definition 6.2.1 (Access)**.** An *access* is a NICQ over $\mathcal{C}$ of the form

$$\hat{R}(x_i, y_i) \leftarrow I(z_i), R(x_i, y_i)$$

where I is a set of *known identifiers*, also called *source*, and either $z_i = x_i$ or $z_i = y_i$. An access induces a new set of known id's: a pattern of the form $\hat{R}(x_i, y_i) \leftarrow I(x_i), R(x_i, y_i)$ induces the new set $\rho(x_i) \leftarrow \hat{R}(x_i, y_i)$ while a pattern of the form $\hat{R}(x_i, y_i) \leftarrow I(y_i), R(x_i, y_i)$ induces the new set $\rho(y_i) \leftarrow \hat{R}(x_i, y_i)$. Set ρ is called the *extraction* of $\hat{R}$. An access $\hat{R}$ inducing an extraction ρ can also be written as

$$I(z_i) \to R(x_i, y_i) \colon \rho(u_i)$$

Definition 6.2.2 (Known identifier)**.** An identifier is *known* iff it belongs to S or if it belongs to the extraction ρ of an access $\hat{R}$.

Definition 6.2.3 (Search plan)**.** A search plan for an admissible query q over $\mathcal{C}$ is a nr-datalog$^\neg$ program P of length n s.t. the first $n-1$ queries are either accesses or extractions and the n-th query, called the *main query* of P, is equivalent to q. The only edb predicates allowed in the n-th query's body are predicates over the seeds $S_j, j \geq 0$[2].

Definition 6.2.4 (Plan cost)**.** Let P be a search plan for an admissible query q. The *cost* of P, denoted by $c(P)$, is the number of distinct accesses of P. In the case of a union of admissible queries $\{q_1, \ldots, q_n\}$, let P_i be a search plan for q_i, $i = 1, \ldots, n$. The cost of the *n-uplet of search plans* $\vec{P} = P_1, \ldots, P_n$, denoted by $c(\vec{P})$, is the number of distinct accesses used in all of $\vec{P}$.

---

[1] We use the $A \to B$ notation to represent the tuples in A as being supplied to extract B

[2] This is to ensure that the R relation is not accessed freely but only via accesses defined earlier in the program.

We are now ready to define the cost-depth of an admissible query.

Definition 6.2.5 (Cost-depth)**.** Let q be an admissible query over $\mathcal{C}$. Let Π be the set of all the search plans for q. The *cost-depth* of q, denoted by $\gamma(q)$ is the following number:

$$\gamma(q) = \min_{P \in \Pi} c(P)$$

Let $q = \{q_1, \ldots, q_n\}$ be a union of admissible queries over $\mathcal{C}$. Let $\vec{\Pi}$ be the set of all the n-uplets of search plans for q, then the *cost-depth* of q is

$$\gamma(q) = \min_{\vec{P} \in \vec{\Pi}} c(\vec{P})$$

6.3 Problem analysis

In this section we present a solution to the problem of finding the cost-depth of an admissible query. We will show how to build a search plan for an admissible query and will discuss minimization issues.

6.3.1 Preliminary examples

We will develop a way to build search plans starting with simple examples and gradually increasing the complexity of the queries.

Example 6.3.1 (Simple query)*.* The first query we examine is the following:

$$q(z) \leftarrow S(x), R_1(x, y), R_2(y, z)$$

We must find a way to answer q by using accesses. We observe that S is used in the query and that it represents a set of known id's. Following the join on x, we supply the x id's in the sender attribute of R_1 to extract all the $\langle x, y \rangle \in R_1$ s.t. $\langle x \rangle \in S$. This gives us access to a new set of identifiers, specifically those represented by y. We can use these id's to access R_2 following the join on y to retrieve all the $\langle y, z \rangle \in R$ s.t. $\langle x, y \rangle \in R$ and $\langle x \rangle \in S$. This strategy has left us with two different fragments of the R relation, one made of the $\langle x, y \rangle$ tuples and the other made of the $\langle y, z \rangle$ tuples. We obtained the two fragments by following the join prescripted by the query itself, so, by projecting onto z, we are able to answer the query. Formally the search plan for q is the following nr-datalog$^\neg$ program:

1. $\hat{R}_1(x, y) \leftarrow S(x), R_1(x, y)$
2. $\rho_1(y) \leftarrow \hat{R}_1(x, y)$
3. $\hat{R}_2(y, z) \leftarrow \rho_1(y), R_2(y, z)$
4. $\hat{q}(z) \leftarrow S(x), \hat{R}_1(x, y), \hat{R}_2(y, z)$

which can be more concisely written as

1. $S(x) \to R_1(x, y)$: $\rho_1(y)$
2. $\rho_1(y) \to R_2(y, z)$

Intuitively, we have replaced every occurrence of the R predicate in q with an appropriate fragment, obtained by accessing R from two distinct sources. The search plan requires two distinct accesses ($\hat{R}_1$ and $\hat{R}_2$) and therefore has a cost of 2. Is there a search plan with smaller cost (1)? The answer is no as we cannot retrieve two different fragments of R with one access.

We now examine an example involving negated predicates, inequalities and where multiple search plans are possible.

Example 6.3.2 (A more complex query). Consider the following query:

$$q(x, y, z) \leftarrow S(x), R_1(x, y), R_2(y, z), S(z), \neg R(x, z), x \neq z$$

Let us first focus on the positive R atoms: we have a total of three different search plans:

P_1:
1. $S(x) \to R_1(x, y)$
2. $S(z) \to R_2(y, z)$

P_2:
1. $S(x) \to R_1(x, y) : \rho_1(y)$
2. $\rho_1(y) \to R_2(y, z) : \rho_2(z)$

P_3:
1. $S(z) \to R_2(y, z) : \rho_2(y)$
2. $\rho_2(y) \to R_1(x, y) : \rho_1(x)$

In all of the above two distinct accesses are involved. For what concerns $\neg R(x, z)$ we must obtain the $\langle x, z\rangle$ tuples and then filter the fragments $\hat{R}_1$ and $\hat{R}_2$ by discarding the tuples s.t. $\langle x, z\rangle \in R$. To achieve this we can either use the accesses $S(x) \to R(x, z)$ or $S(z) \to R(x, z)$ in all search plans, $\rho_2(z) \to R(x, z)$ in P_2 or $\rho_1(x) \to R(x, z)$ in P_3. For every search plan we now discuss which is the best way to obtain the $\langle x, z\rangle$ tuples. In plan P_1, it is equivalent to use access pattern $S(x) \to R(x, z)$ or $S(z) \to R(x, z)$ since the former is equivalent to $\hat{R}_1$ and the latter is equivalent to $\hat{R}_2$, both already in P_1, and therefore the cost of the whole search plan is still 2 in both cases. In P_2, the access $S(x) \to R(x, z)$ is also equivalent to $\hat{R}_1$ and thus the plan cost does not change while, if we chose $\rho_2(z) \to R(x, z)$, we would be using a new access and raise the cost of the plan to 3. Finally, in P_3, for the same reason, the best way to obtain the $\langle x, z\rangle$ tuples is to use the access $S(z) \to R(x, z)$, which is equivalent to $\hat{R}_2$. At this point we can conclude that the three search plans have the same cost, even considering the negated atom. Last, we address the $x \neq z$ atom. In general, to evaluate inequalities, access to R is not necessary since it only suffices to filter the answer of the query without inequalities by comparing pairs of *already obtained* constants. This implies that inequality predicates have no impact on cost-depth evaluation. We now show, to conclude the example, the full nr-datalog$^\neg$ program of P_1:

1. $\hat{R}_1(x, y) \leftarrow S(x), R_1(x, y)$
2. $\hat{R}_2(y, z) \leftarrow S(z), R_2(y, z)$
3. $\hat{q}(x, y, z) \leftarrow S(x), \hat{R}_1(x, y), \hat{R}_2(y, z), S(z), \neg\hat{R}_1(x, z), x \neq z$

In the last query we have written $\neg\hat{R}_1(x, z)$ in place of $\neg R(x, z)$ because access $S(x) \rightarrow R(x, z)$, which retrieves the $\langle x, z\rangle$ tuples, is equivalent to $\hat{R}_1$.

6.3.2 Building a search plan

Given the definitions in Section 6.2 and the observations in Section 6.3.1, we now provide a strategy toward search plan definition. Given an admissible query q over $\mathcal{C}$

1. to each $R_j(x_j, y_j)$ atom we associate an access where the source is a seed S_k joined with R_j or the extraction $\rho_i(z_i)$ of an access $\hat{R}_i$ s.t. R_i is joined with R_j on z_i;
2. to each $\neg R_j(x_j, y_j)$ atom we associate an access $I(z_i) \rightarrow R(x_i, y_i)$ s.t. the source I contains either all the $\langle x_j\rangle$ tuples or all the $\langle y_j\rangle$ tuples (as done in example 6.3.2). This ensures that all the $\langle x_j, y_j\rangle$ tuples are extracted and can be used to filter the answer as if there were no access limitations;
3. The main query of the search plan has the same head as q and every $R_j(x_j, y_j)$ atom is replaced by the associated access;
4. the cost of the search plan is computed as the number of distinct accesses used, where distinct means *not equivalent queries*.

This strategy ensures that the main query of a search plan P for q returns the same answer as q because by undoing the replacing of R atoms with $\hat{R}$ accesses we obtain the original query: for instance, in example 6.3.2, if we "expand" all the accesses in the main query

$$\hat{q}(x, y, z) \leftarrow S(x), \hat{R}_1(x, y), \hat{R}_2(y, z), S(z), \neg\hat{R}_1(x, z), x \neq z$$

we obtain

$$\hat{q}(x, y, z) \leftarrow S(x), R_1(x, y), R_2(y, z), \neg\hat{R}_1(x, z), x \neq z$$

Query $\hat{q}$ is equivalent to q because they contain the same positive atoms and $\hat{R}_1$ contains all the $\langle x, z\rangle$ tuples.

6.3.3 Importance of minimization

We now show the effects of minimization on the cost-depth evaluation.

Example 6.3.3 (Cost-depth and query minimization). Consider query q:

$$q(x, z, u) \leftarrow S(x), S(u), R_1(x, y), R_2(z, x), R_3(x, u)$$

For this query a possible search plan is

P: 1. $\hat{R}_1(x,y) \leftarrow R_1(x,y), S(x)$
2. $\hat{R}_2(z,x) \leftarrow R_2(z,x), S(x)$
3. $\hat{R}_3(x,u) \leftarrow R_3(x,u), S(u)$
4. $\hat{q}(x,z,u) \leftarrow S(x), S(u), \hat{R}_1(x,y), \hat{R}_2(z,x), \hat{R}_3(x,u)$

We have that $c(P) = 2$ ($\hat{R}_2 \equiv \hat{R}_3$) and there are no search plans for q with smaller cost than P. If atom $R_1(x,y)$ is removed we obtain the minimal query $q_m \equiv q$[1]:

$$q_m(x,z,u) \leftarrow S(x), S(u), R_1(z,x), R_2(x,u)$$

q_m admits the following search plan:

P_m: 1. $\hat{R}_1(z,x) \leftarrow R_1(z,x), S(x)$
2. $\hat{R}_2(x,u) \leftarrow R_2(x,u), S(u)$
3. $\hat{q_m}(x,z,u) \leftarrow S(x), S(u), \hat{R}_1(z,x), \hat{R}_2(x,u)$

Clearly $c(P_m) = 1$ ($\hat{R}_1 \equiv \hat{R}_2$) and thus we can conclude that $\gamma(q) = 1$, since a query with at least one R-atom cannot admit a search plan with no accesses.

The above example shows that minimization can reduce the number of accesses required to answer the query.

6.3.4 Cost-depth and nr-datalog$^\neg$ programs

We conclude this section by addressing how to evaluate the cost-depth of a query q when q is part of a nr-datalog$^\neg$ program. In this case the only difference is the presence of idb predicates. We assume all the queries in the program to be unary and that the cost-depth of the previous queries q_i has been already evaluated. This means that they already have an associated search plan. We can then consider the identifiers returned by every q_i to be *known*. These considerations have the following consequences:

- idb atoms of the form $q_i(x_i)$ can be used as access sources;
- when evaluating the cost of a search plan for q_j, one must take into account all the accesses used directly in q_j and all the accesses used in the queries q_i, $i < j$, s.t. $q_i(x_i)$ or $\neg q_i(x_i)$ appear in the body of q_j.

Example 6.3.4 (Cost-depth and nr-datalog$^\neg$ programs). Consider the following nr-datalog$^\neg$ program Q:

1. $q_1(y) \leftarrow S(x), R(x,y)$

[1] Intuitively $R_1(x,y)$ states the same concept as $R_3(x,u)$. The latter is also joined with $S(u)$ making it more specific so if a tuple matches $R_3(x,u), S(u)$ it also matches $R_1(x,y)$ which is redundant.

2. $q_2(y) \leftarrow q_1(x), R(x,y), \neg q_1(y)$

The only search plan for q_1 solely relies on the access $S(x) \rightarrow R(x,y)$. Likewise, the only search plan for q_2 consists of the access $q_1(x) \rightarrow R(x,y)$[1]. Summing up, two distinct accesses are required to answer q_2 hence $\gamma(q_2) = 2$.

6.3.5 Problem complexity

Before developing algorithms for cost-depth computation, it is critical to understand the complexity of the problem. At first glance cost-depth computation is an optimization problem, where we must find the minimum of the cost function c over all $P \in \Pi$, where Π contains all the possible search plans for the target query q. We now provide some background on a very well known and studied optimization problem known as the *minimum set cover problem*.

Definition 6.3.1 (Minimum set cover problem (MSC))**.** Let $I = \{1, \ldots, n\}$ be a set of elements and let $C \subseteq 2^I$ be a set of subsets of I. The goal of the minimum set cover problem, given I and C, is to find the smallest $P \subseteq C$ such that $\bigcup_{P_i \in P} P_i \equiv I$. More formally, the goal is to find

$$\min_{P:\ \bigcup_{P_i \in P} P_i \equiv I} |P|$$

The minimum set cover problem has been proven to be NP-hard [17]. We now make the following claim:

Theorem 6.3.1 (NP-hardness of cost-depth computation (CDC))**.** *The problem to compute the cost-depth of an admissible query over $\mathcal{C}$ is NP-hard.*

Proof. To prove that CDC is NP-hard we now present an polynomial reduction procedure to transform an instance of the minimum set cover problem into an instance of the cost-depth computation problem. Let (I, C) be an instance of the MSC, where $C \subseteq 2^I$. The procedure is the following:

1. for each $C_j \in C$ define the seed S_j;
2. let q be a NICQ over $\mathcal{C}$ with empty head and body;
3. for each $i \in I$ add atom $R(x_i, y_i)$ to q's body;
4. for each C_j and $i \in C_j$ add atom $S_j(x_i)$ to q's body;
5. add every variable in q's body to q's head.

[1] Atom $\neg q_1(y)$ does not require any additional access since q_1's output is known

The procedure has a complexity of $\Theta(|C| + |I| + \sum_{j=1}^{|C|} |C_j|) = O(|C| + |I| + |C| \times |I|)$ hence it is polynomial. We now show that the procedure actually reduces MSC to CDC. We first point out that to each $i \in I$ we associate atom $R(x_i, y_i)$ and to each C_j we associate a seed S_j. Moreover an S_j-atom and $R(x_i, y_i)$ atom are joined iff $i \in C_j$. Let us consider the solution of the CDC instance: it consists of the search plan requiring the minimum number of accesses. In the considered query all accesses are of the form $S_j(x_i) \rightarrow R(x_i, y_i)$ because there are no joins between different R-atoms. The found search plan can be therefore seen as the *best subset* of seeds S_j accessing all R-atoms. Since all S_j are only joined with the sender attribute of some R-atoms, to each S_j is associated only one access. It is clear that the solution to the CDC problem corresponds to the solution of the starting minimum set cover problem: the best S_j seeds/accesses to cover all the R-atoms correspond to the best set of subsets C_j covering all $i \in I$.

Example 6.3.5 (Reduction from MSC to CDC). Let $I = \{1, 2, 3, 4, 5\}$ and $C = \{\{1\}, \{2, 3\}, \{4, 5\}, \{2, 3, 4, 5\}\}$. The best subset of C covering I is obviously $\{\{1\}, \{2, 3, 4, 5\}\}$. By applying the reduction procedure we obtain the following query:

$$\begin{aligned} q(x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4, x_5, y_5) \leftarrow & R(x_1, y_1), R(x_2, y_2), R(x_3, y_3), R(x_4, y_4), R(x_5, y_5), \\ & S_1(x_1), S_2(x_2), S_2(x_3), S_3(x_4), S_3(x_5), \\ & S_4(x_2), S_4(x_3), S_4(x_4), S_4(x_5) \end{aligned}$$

The best search plan for q is

P:
1. $S_1(x_1) \rightarrow R(x_1, y_1)$
2. $S_4(x_2) \rightarrow R(x_2, y_2)$
3. $S_4(x_3) \rightarrow R(x_3, y_3)$
4. $S_4(x_4) \rightarrow R(x_4, y_4)$
5. $S_4(x_5) \rightarrow R(x_5, y_5)$

and its cost is $c(P) = \gamma(q) = 2$.

□

Having proven that cost-depth computation is NP-hard, we expect it to be intractable for complex queries. We also find it hard, although we have not been able to prove it, that CDC is in NP, mainly because the existence of a polynomial procedure to verify a solution to the problem seems unlikely. In light of this discovery on the problem complexity, in the next section we will both provide an exact algorithm and a heuristic algorithm to compute the cost-depth of an admissible query.

6.4 Proposed algorithms

In this section we propose two algorithms to find the cost-depth of an admissible query. Since we have proven the problem to be NP-hard, we will present an exact *branch and bound* algorithm and an approximated *greedy* algorithm. Before presenting the two algorithms, we will describe the data structure used to represent candidate search plans.

6.4.1 Search graph

Given an admissible query q over $\mathcal{C}$, we want to find the search plan P minimizing $c(P)$. In order to do so we introduce the *search graph* as a way to represent all the possible accesses that can be used to answer the query.

Definition 6.4.1 (Search graph)**.** Let q be an admissible query over $\mathcal{C}$. The *search graph* of q is the arc-labeled multidigraph AG_q such that

1. for every atom $S_j(z_j)$ ($q_j(z_j)$) in q there is a unique node S_j (q_j) in AG_q;
2. for every atom $R_j(x_j, y_j)$ in q there is a node R_j in AG_q;
3. for every atom $S_j(z_j)$ ($q_j(z_j)$) and $R_i(x_i, y_i)$ in q joined via variable t_k, there is an arc (S_j, R_i, t_k) ((q_j, R_i, t_k));
4. for every $R_j(x_j, y_j)$ and $R_i(x_i, y_i)$ in q, joined via variable t_k, then arcs (R_i, R_j, t_k) and (R_j, R_i, t_k) are in AG_q;
5. nothing else belongs to AG_q.

The search graph has a very intuitive meaning: an arc (i, j, ℓ) in AG_q represents the fact that the extraction of the access identified by i can be used to define an access for the atom identified by j via the variable ℓ. Specifically an arc (R_i, R_j, t_k) encodes the fact that the extraction ρ_i obtained by accessing $R_i(x_i, y_i)$ can be used to access $R_j(x_j, y_j)$ via the common variable t_k. Therefore to find a search plan P for q is equivalent to find exactly one incoming arc to every R_i node such that every R_i can be reached from some S_j or q_j node.

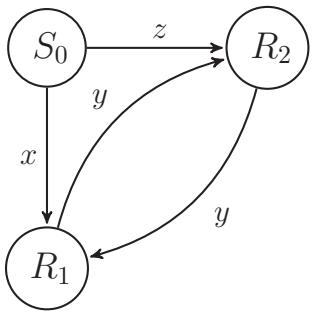


Figure 6: Search graph for query $q(x, y, z) \leftarrow S_0(x), S_0(z), R_1(x, y), R_2(y, z)$.

Whenever an arc is *chosen*, some other arcs must be removed from the graph, e.g. in Figure 6, if we choose arc (S_0, R_1, x), arc (R_2, R_1, y) is removed because R_1 is already accessed via S_0. An algorithm with the aim to find the cost-depth of q will repeatedly choose certain arcs in AG_q until no more arcs can be added to the solution. We now present procedure *PruneGraph* which, given a search graph AG_q and a newly chosen arc (i, j, ℓ), removes all the arcs that must be excluded following the choice of (i, j, ℓ).

Algorithm 2: *PruneGraph*

Input: a search graph AG_q and an arc $(i, j, \ell) \in AG_q$
Output: the pruned search graph ${AG_q}'$

```
1 AG_q' ← AG_q
2 for (i', j', ℓ') ∈ AG_q'.In(j) \ {(i, j, ℓ)} do
3 |   AG_q' ← AG_q' \ {(i', j', ℓ')}
4 end
5 for (i', j', ℓ') ∈ AG_q'.Out(j) do
6 |   if ℓ = ℓ' ∨ j' = i then
7 |   |   AG_q' ← AG_q' \ {(i', j', ℓ')}
8 end
9 return AG_q'
```

The first **for** loop removes all the incoming arcs of j because we have chosen (i, j, ℓ) to access it. The second **for** loop removes all the outgoing arcs of j having the same variable

ℓ that was used to access j in the first place[1] and any arc from j to i if they exist. The time complexity of `PruneGraph` is linear in the number of arcs in AG_q.

6.4.2 Branch and bound search plan finder

The first algorithm we discuss is a *branch and bound* (B&B) algorithm. B&B algorithms are employed in combinatorial optimization when the growth of the size of the problem makes brute force enumeration of the possible solutions not computationally feasible. A B&B algorithm finds the optimal solution to a problem by building the solution step by step. At each step, several alternatives called branches are considered. *For each branch* the following decision is made: a branch is not taken if the solution resulting in taking the branch is estimated not to be better than the best solution found so far, called *bound*. If otherwise the branch is estimated to lead to a possibly better solution than the bound, the branch is taken. The search performed by a B&B algorithm can be then represented by a tree. B&B algorithms are guaranteed to find the optimal solution to any combinatorial optimization problem, provided that the bounds are chosen correctly.

Algorithm `BranchAndBoundPlanFinder` receives an admissible query q_j over $\mathcal{C}$ and returns the value of $\gamma(q_j)$. To accomplish this goal, first q_j is minimized then the search graph AG for q_j is built according to def. 6.4.1. Then variable c^*, representing the cost of the best plan so far, is set to ∞. We then scan the body of q_j for idb atoms. For every idb atom $(\neg)q_i(x_i)$, $i < j$, for which cost-depth has been already computed, we add all the accesses made in q_i ($Accesses[q_i]$) to the accesses made in q_j ($Accesses[q_j]$). The integer c is updated to $|Accesses[q_j]|$. Procedure `BranchAndBoundRecursive`, which updates c^* to the actual $\gamma(q)$ is then called [2]. Finally, `BranchAndBoundPlanFinder` returns c^*.

Algorithm `BranchAndBoundRecursive` receives in input a search graph AG representing the available search space so far, a list of arcs called $Available$ representing the arcs that can be chosen currently (accesses s.t. the required extraction has been already chosen), a set of nodes called $Chosen$ which keeps track of the accessed atoms so far, the set $Accesses[q_j]$ containing the accesses made so far, and an integer c representing the current cost of the search plan. The algorithm updates variable c^* several time during execution. Once there are no more arcs to choose from, procedure `BranchAndBoundNegated` is called to evaluate the additional cost of accessing the negated R atoms in q_j.

Procedure `RemoveFirst`$(list)$ removes and returns the first element of the list passed as argument. Procedure `AddAll`$(list, set)$ adds all the elements of set at the end of $list$ while procedure `RemoveAll`$(list, set)$ removes all the elements of set from $list$.

---

[1] There is no point to perform the access $S(x) \rightarrow R_1(x, y) : \rho_1(x)$ and then $\rho_1(x) \rightarrow R_2(z, x)$ since $S(x) \rightarrow R_2(z, x)$ will do it as well. This is why we have only allowed extractions of non accessed variables (e.g. y).

[2] We assume that function parameters are always passed by value.

Algorithm 3: *BranchAndBoundPlanFinder*

Input: An admissible query q_j over $\mathcal{C}$
Output: An integer c^* with value $\gamma(q_j)$

```
1  q_j ← the minimal query equivalent to q_j
2  AG: the search graph of q_j
3  Available: a list containing all the arcs (S_j, R_k, z_i), (q_j, R_k, z_i) ∈ AG
4  Chosen: an empty set of nodes
5  Accesses[q_j]: an empty set of accesses
6  Accesses*[q_j]: an empty set of accesses
7  c* ← ∞
8  for every idb atom (¬)q_i(x_i) in q_j's body do
9  |  Accesses[q_j] ← Accesses[q_j] ∪ Accesses[q_i]
10 end
11 c ← |Accesses[q_j]|
12 BranchAndBoundRecursive (AG, Available, Chosen, Accesses[q_j], c)
13 Accesses[q_j] ← Accesses*[q_j]
14 return c*
```

As the name suggests, `BranchAndBoundRecursive` is a recursive algorithm: it removes the first arc from $Available$ and explores two possibilities, to skip to the next arc in $Available$ by simply calling `BranchAndBoundRecursive` or to add the arc to $Chosen$, prune the graph, update $Available$ and the cost of the solution and calling `BranchAndBoundRecursive`. The first branch is taken only if by removing the current arc there is at least another available arc to access node j (line 6). This check forbids the exploration of unfeasible branches. The second branch is taken only is the updated cost c' is smaller than c^* (line 11). This last check prevents the algorithm from exploring solutions which are not better than the currently best solution [1]. The base case of the algorithm (line 1) is reached when there are no more arcs in $Available$ and all nodes R_i of AG are in $Complete$. If the above hold, the algorithm calls `BranchANdBoundNegated` and returns to the caller.

Algorithm `BranchAndBoundNegated` (algorithm 5) receives the access set built so far ($Accesses[q_j]$), a list containing all the negated R atoms left to address ($Available$) and the cost of the plan so far (c). Being this a branch & bound algorithm as before, we end when we have no more negated R atoms to deal with and we explore the possible solutions by branching. The main difference is that this time we are branching on the possible accesses

[1] From the B&B paradigm, c^* is the upper bound to the solution cost.

Algorithm 4: *BranchAndBoundRecursive*

Input: A search graph AG, a list of arcs $Available$, a set of nodes $Chosen$, a set of accesses $Accesses[q_j]$ and an integer c
Output: none

if $Available = \emptyset$ **then**
  if $Chosen$ *contains all the* R_i *nodes of* AG **then**
    `BranchAndBoundNegated` ($Accesses[q_j]$, all negated R atoms in q_j's body, c)
    return
$(i, j, \ell) \leftarrow$ `removeFirst`$(Available)$
if $|AG.In(j)| > 1$ **then**
  `BranchAndBoundRecursive` $(AG, Available, Chosen, Accesses, c)$
$AG' \leftarrow$ `PruneGraph`$(AG, (i, j, \ell))$
$Chosen' \leftarrow Chosen \cup \{j\}$
$Available'$: an empty list of arcs
if (i, j, ℓ) *is not in* $Accesses[q_j]$ **then**
  $c' \leftarrow c + 1$
$Accesses'[q_j] \leftarrow Accesses[q_j] \cup \{$the access encoded by $(i, j, \ell)\}$
if $c' < c^*$ **then**
  `addAll`$(Available', \{(i', j', \ell') \,|\, (i', j', \ell') \in AG.Out(j) \wedge j' \notin Chosen\})$
  `removeAll`$(Available', \{(i', j', \ell') \,|\, (i', j', \ell') \notin AG\})$
  `BranchAndBoundRecursive` $(AG', Available', Chosen', Accesses', c')$
return

available for each atom $\neg R_i(x_i, y_i)$ in the query. As we have seen in Section 6.3.2, if we found an access containing all the tuples involved in $\neg R_i(x_i, y_i)$, we could use such an access to filter the answer, discarding those tuples s.t. $\langle x_i, y_i \rangle \in R$. Such an access $\hat{R}'$ has been accessed from a source I and a sufficient condition for this access to contain all the $\langle x_i, y_i \rangle$ tuples is that $q_{x_i} \subseteq I$ if $I(x_k) \rightarrow R'(x_k, y_k)$ and that $q_{y_i} \subseteq I$ if $I(y_k) \rightarrow R'(x_k, y_k)$, where q_z is the query having all the positive atoms of q in its body and only the variable z in the head. Let us examine the former condition (the latter is symmetric): q_{x_i} outputs all and only the constants that appear in the x_i field of the answer to the full query q and therefore exactly those involved in the $\neg R(x_i, y_i)$ filtering. If $q_{x_i} \subseteq I$ holds then all such constants are in I and therefore access $I(x_k) \rightarrow R'(x_k, y_k)$ contains all the $\langle x_i, y_i \rangle$ tuples. If this holds for some access $\hat{R}'$ then there is no additional cost associated to atom $\neg R(x_i, y_i)$. We point out that to establish this fact, it is required to perform containment checking of

q_{x_i} and q_{y_i} with potentially every source used in P [1]. The above checks are performed at lines 6-13.

If we cannot find "compatible" accesses we can simply create a new access from a source with some variables in common with $\neg R_i(x_i, y_i)$. For instance, in query $q(y) \leftarrow S(x), R(x, y), \neg R(y, z), S(z)$, where the only possibility to access the positive R-atom is $S(x) \rightarrow R(x, y) : \rho(y)$, we can access the negated atom either with $\rho(y) \rightarrow R(x, z)$ or with $S(z) \rightarrow R(x, z)$. In any case, the cost-depth of q is 2. This operation requires to potentially check every access made so far, looking for common variables. This computation is performed at lines 14-19. As in `BranchAndBoundRecursive` we cut branches that are guaranteed not to improve on the current solution (line 17).

Time complexity of *BranchAndBoundPlanFinder*

Up to line 7 of `BranchAndBoundPlanFinder` the complexity of the algorithm is dominated by the minimization at line 1 which, as pointed out in previous chapters, is Π_2^p-complete. The **for** loop at lines 8-10 scans the body of q_j and adds two sets at every iteration, therefore it has a time complexity of $O(p_j + (j-1) \times A)$ where p_j is the number of atoms in q_j's body and A is the maximum number of accesses required for $q_1, \ldots, q_{j-1}$. It is reasonable to assume this cost negligible w.r.t. the minimization procedure. Now, for the core of the algorithm, `BranchAndBoundRecursive` is a branch and bound algorithm and therefore its time complexity coincides, in the worst case scenario, with the one of the exhaustive search. The algorithm explores the search space by considering each arc and creating a first subproblem where the arc is discarded and a second subproblem where the arc is part of the solution. Since a search plan involves exactly r arcs of AG, where r is the number of R-atoms in q_j, the algorithm will explore a number of solutions not greater than $\binom{|A|}{r}$ where A is the arc set of AG. Let n be the number of seeds in the schema (included S_0) and let v be the number of variables in q_j. Atom $S_j(x_k)$ can appear v times in q_j and each $S_j(x_j)$ atom can be joined with $v-1$ atoms of the form $R(x_j, y_j)$ and $v-1$ atoms of the form $R(y_j, x_j)$. In total there can be $n \times v \times 2(v-1)$ arcs from seeds to R nodes. Via a similar reasoning we can conclude that there are at most $(j-1) \times v \times 2(v-1)$ arcs from idb nodes to R nodes and at most $v \times (v-1) \times (2v-3)$ arcs from R nodes to R nodes. It follows that $|A| \leq (2n + 2j + 2v - 5)v(v-1)$. Now we have that

$$\binom{|A|}{r} = \frac{|A|!}{r!(|A|-r)!} = \frac{\prod_{k=0}^{r+1}(|A|-k)}{r!} \leq \frac{|A|^{r+2}}{r!} = O(\frac{(nv^2 + jv^2 + v^3)^{r+2}}{r!})$$

We can observe that the above number grows polinomially as n, v and j increase but its behavior as a function of r depends on the size of the numerator [2]. All the helper

[1] Containment for NICQs is Π_2^p complete.

[2] $r!$ is both $\omega(2^r)$ and $o(r^r)$

procedures used in `BranchAndBoundRecursive` have linear complexity. Let us not forget that, when reaching its base case, `BranchAndBoundNegated` is called: each call to `BranchAndBoundNegated` can generate up to $r + e$ recursive calls because of the *compatible* accesses already in the plan (lines 6-13) and up to r recursive calls because of the joined atoms (lines 14-19). Since $Available$ is always reduced of one element, `BranchAndBoundNegated` can expand up to $(e \times (2r + e))^e$ leaves. We also point out with each call to `BranchAndBoundNegated`, up to $e \times (r + e)$ containment tests for conjunctive queries need to be performed (lines 7, 11), which require solving an NP-complete problem [12, p. 120-121]. In Chapter 8 we will see that the execution time of the algorithm grows drastically even with small increases of the number of variables v.

6.4.3 Greedy plan finder

We now present a *greedy* algorithm to find the cost-depth of a query. Greedy algorithms are employed in combinatorial optimization problems to find a solution which is good but not necessarily the optimal one. To achieve this, a greedy algorithm builds the solution step by step but, at each step, the next element to add to the solution is chosen following some criteria of *local optimality*. Choices cannot be undone ad once the choice is made, the algorithm goes to the next step. Since no backtracking is involved, greedy algorithms have usually faster running times than B&B algorithms. Algorithm `GreedyPlanFinder` (algorithm 6) receives in input an admissible query q over $\mathcal{C}$ and finds a search plan P for q, which is not necessarily the one with minimum cost, and returns $c(P)$. Because of this approximation, `GreedyPlanFinder` generally returns an *upper bound to* $\gamma(q)$.

As in `BranchAndBoundRecursive`, $Available$ is a list containing all the arcs of AG that can be added to the solution. We point out that `GreedyPlanFinder` does not minimize the input query because the goal is to obtain the results quickly. The algorithm adds the accesses made by previous queries to $Accesses[q_j]$, as done in `BranchAndBoundPlanFinder`, then it iteratively compares all the arcs in $Available$ and chooses the one bringing the smallest increment to the cost of the search plan so far. The `while` loop is exited when $Available$ is empty. At line 11, procedure `shuffle` randomly rearranges the elements of $Available$. This is done because the cost-depth estimate obtained can be affected by the order of the arcs in $Available$. The reason for this behavior is inherent in the mechanism of every Greedy algorithm: the algorithm finds a solution by searching for local optima which are not guaranteed to provide the global optimum. Shuffling $Available$ leads to a non deterministic behavior of the algorithm therefore a clever use of the algorithm would be to run `GreedyPlanFinder` a certain number of times and then take the smallest obtained value of Γ. After choosing an access for every positive R-atom the algorithm provides an access for every negated R-atom (lines 25-30). In order to do so we check for the following special case of containment: if in access $I(x_k) \rightarrow R'(x_k, y_k)$ ($I(y_k) \rightarrow R'(x_k, y_k)$), $x_k = x_i$ ($y_k = y_i$), we automatically have that $q_{x_i} \subseteq I$ ($q_{y_i} \subseteq I$) because I has the same head variable x_i (y_i) as q_{x_i} (q_{y_i}) and, from how sources are defined, all the atoms in I are in q_{x_i} (q_{y_i}). If

this holds then the current negated R-atom does not require an extra access, otherwise we add a new access raising Γ by 1, just as done at lines 14-19 of `BranchAndBoundNegated`.

We provide an example to show that, in general, `GreedyPlanFinder` does not find the real cost-depth $\gamma(q)$.

Example 6.4.1 (Greedy algorithm and cost-depth). Consider query q:

$$q(x, y, z, t, u) \leftarrow S_1(x), S_1(z), S_2(u), S_2(u'), R_1(x, y), R_2(z, u), R_3(y, v), R_4(u, w), R_5(t, u')$$

search graph AG_q is shown in Figure 7. Suppose `GreedyPlanFinder`(q) is executed. At the beginning $Available$ contains arcs $(S_1, R_1, x), (S_1, R_2, z), (S_2, R_5, u), (S_2, R_2, u)$ and (S_2, R_4, u). Suppose (S_2, R_5, u) is picked first with cost 1. The **for** loop will keep this choice as all first accesses cost 1. Of all accesses considered in the second iteration of the **while** loop, the best one is (S_2, R_2, u) since it is a variant of (S_2, R_5, u') and therefore costs 0. This way, the best plan we can obtain features arcs (S_1, R_1, x), (R_1, R_3, y), which are mandatory, and one among (R_2, R_4, u) and (S_2, R_4, u) since they both cost 1. Either way, the overall cost of the search plan is 4. This is not the best search plan that can be obtained since the search plan consisting of arcs (S_1, R_1, x), (R_1, R_3, y), (S_1, R_2, z), (R_2, R_4, u) and (S_2, R_5, u') has a cost of 3[1].

Time complexity of *GreedyPlanFinder*

Building the search graph has a complexity of $O(p_j^2)$ where p_j is the number of atoms in q_j's body. The call to `addAll` takes a time $O(|A|)$ where A is the arc set of AG. The `for` loop at lines 4-6 takes $O(p_j + (j-1) \times C)$ iterations as in `BranchAndBoundPlanFinder`, where C is the maximum number of accesses for $q_1, \ldots, q_{j-1}$. The `while` loop (lines 9-24) runs for r iterations, where r is the number of positive R-atoms in q_j because we must pick exactly r arcs and an arc is chosen at every iteration. The `for` loop (lines 12-17) scans $Available$ for the best arc and therefore requires $O(|A|)$ iterations. Every operation in its body has constant execution time. The operations at lines 18-23 have an overall complexity of $O(|A|)$. In total, the **while** loop has a complexity of $O(r \times |A|)$. The last `for` loop (lines 25-30) runs for e iterations, where e is the number of negated R-atoms in q_j's body. The **if** condition at line 26 can be checked by scanning A and therefore takes a time of $O(|A|)$. The same can be said for line 27. Based on the above, the last **for** loop takes $O(e \times |A|)$ iterations. It is evident that the running time of the overall algorithm, which is $O(p_j^2 + (j-1)C + (r+e)|A|)$, grows as a polynomial of the input size. In Chapter 8 we will evaluate the running time of `GreedyPlanFider` by performing experimental trials on randomly generated queries. This will serve both to evaluate the time performance of `GreedyPlanFinder` and to see how close the estimated value Γ is to the real $\gamma(q)$.

---

[1]The accesses encoded by (S_1, R_1, x) and (S_1, R_2, z) are equivalent and so are (R_1, R_3, y) and (R_2, R_4, u).

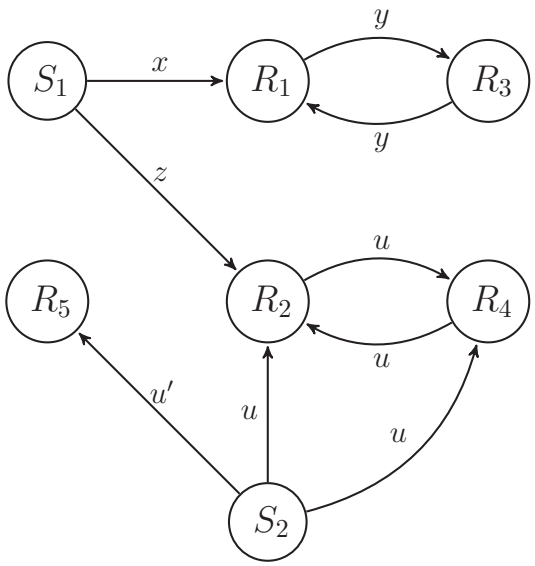


Figure 7: Search graph for example 6.4.1.

6.5 Comparison with the graph-depth approach

We now compare the graph-depth and the cost-depth of a query. We first show that the two notions are related, then we discuss the pros and cons of each approach.

6.5.1 Relation between the two approaches

Here we shall prove that, given an admissible query q over $\mathcal{C}$, the relation $\delta(q) \leq \gamma(q)$ holds. The proof is complex so we have broken it in the following steps:

1. prove that $\delta(q) \leq \gamma(q)$ holds for *conjunctive queries* (CQ)[1];
2. prove that the above claim still holds when adding negation and inequalities;
3. prove that the above claim holds for UNICQs;
4. prove that the above claim holds for a generic q_j in a nr-datalog$^\neg$ program.

Proof of step 1. To prove that for an admissible CQ q over $\mathcal{C}$ $\delta(q) \leq \gamma(q)$ we will

1.a show a CQ q_1 for which $\delta(q_1) < \gamma(q_1)$;

1.b show a CQ q_2 for which $\delta(q_2) = \gamma(q_2)$;

[1] CQs are NICQs without negated atoms and inequalities.

1.c prove that $\delta(q) > \gamma(q)$ is impossible.

Step 1.a: consider CQ $q_1(x) \leftarrow S(x), R_1(x,y), R_2(z,x)$. From the call graph in Figure 8 it follows that $\delta(q_1) = 1$. The query is minimal and the best search plan for q_1 is:

P_1:
1. $\hat{R}_1(x,y) \leftarrow S(x), R_1(x,y)$
2. $\hat{R}_2(z,x) \leftarrow S(x), R_2(z,x)$
3. $\hat{q}_1(x) \leftarrow S(x), \hat{R}_1(x,y), \hat{R}_2(z,x)$

Since $c(P_1) = 2$ then $\gamma(q_1) = 2$.

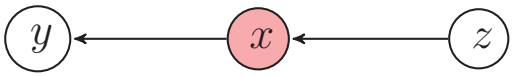


Figure 8: Call graph for step 1 of the proof.

Step 1.b: consider query $q_2(z) \leftarrow S(x), R_1(x,y), R_2(y,z)$. From the call graph in Figure 9 it follows that $\delta(q_2) = 2$. The query is minimal and the best search plan for q_2 is:

P_2:
1. $\hat{R}_1(x,y) \leftarrow S(x), R_1(x,y)$
2. $\rho_1(y) \leftarrow \hat{R}_1(x,y)$
3. $\hat{R}_2(y,z) \leftarrow \rho_1(y), R_2(y,z)$
4. $\hat{q}_2(z) \leftarrow S(x), \hat{R}_1(x,y), \hat{R}_2(y,z)$

Since $c(P_2) = 2$ then $\gamma(q_2) = 2$.

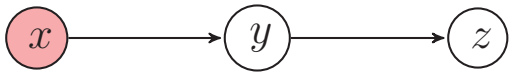


Figure 9: Call graph for step 2 of the proof.

Step 1.c: suppose that, for an admissible CQ q over $\mathcal{C}$ of the form $q(\vec{X}) \leftarrow conj(\vec{X}, \vec{Y})$, $\delta(q) = k > 0$ and $\gamma(q) = k' < k$. Since $\delta(q) = k$ there must be atoms $S_k(z), R_{j_1}(x_{j_1}, y_{j_1}), \ldots, R_{j_k}(x_{j_k}, y_{j_k})$ in q such that $S_k(z)$ and $R_{j_1}(x_{j_1}, y_{j_1})$ are joined, $R_{j_i}(x_{j_i}, y_{j_i})$ and $R_{j_{i+1}}(x_{j_{i+1}}, y_{j_{i+1}})$ are joined for $i = 1, \ldots, k-1$ and $S_k(z_{k,h})$ is the closest seed atom, in terms of the number of joins involved, to any of the $R_{j_i}(x_{j_i}, y_{j_i})$. Figure 10 shows the portion of the call graph G_q representing these atoms. Some arcs in the figure are labeled with the atoms of q they represent and we ignore the orientation of the arcs. A dashed arc between nodes i and j represents a chain between the two nodes.

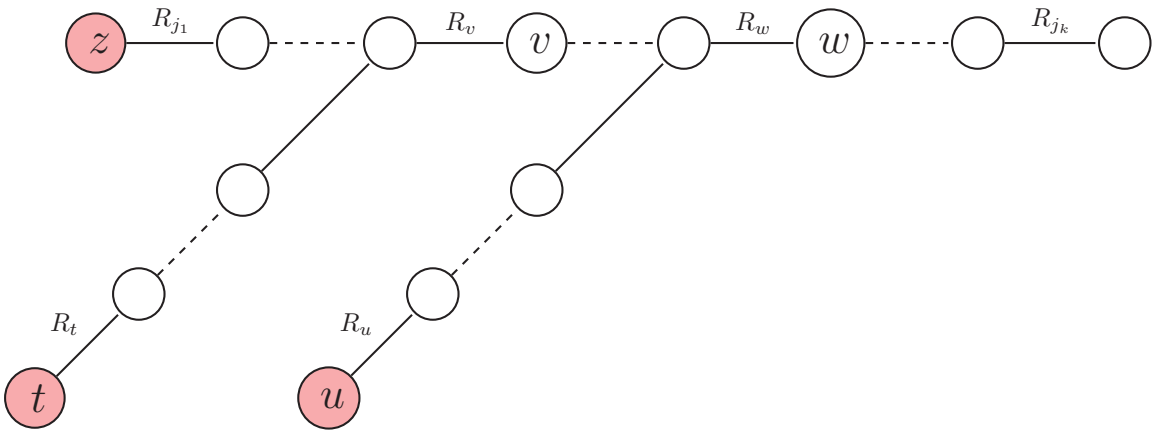


Figure 10: Call graph for step 3 of the proof.

For $\gamma(q) = k' < k$ to hold, the cost of the best search plan P for q is $c(P) = k'$. Since $c(P)$ is the number of accesses to R that P performs, it must be that some of the $\hat{R}_{j_1}, \ldots, \hat{R}_{j_k}$ are equivalent. Consider arcs labeled with R_v and R_w in Figure and suppose that $\hat{R}_v$ is equivalent to $\hat{R}_w$. This means that every access in the sequence $\hat{R}_t, \ldots, \hat{R}_v$ is equivalent to the respective access in $\hat{R}_u, \ldots, \hat{R}_w$. Let $l(i, j)$ denote the length of a shortest chain between nodes i and j. Since $\delta(q) = k$, we have that $l(t, v) \geq l(z, v)$ and $l(u, w) \geq l(z, w)$. Then, since $\hat{R}_v$ is a variant of $\hat{R}_w$, $l(t, v) = l(u, w)$. But this is impossible because $l(z, v) < l(z, w)$ and therefore $l(t, v) < l(u, w)$. If follows that there are no equivalent accesses among $\hat{R}_{j_1}, \ldots, \hat{R}_{j_k}$. This proves that $\delta(q) > \gamma(q)$ is impossible. □

Proof of step 2. At this point is is obvious that $\delta(q) \leq \gamma(q)$ holds for full NICQs because

- inequalities do not affect neither $\delta(q)$ nor $\gamma(q)$;
- negated R-atoms do not add hops to a query but they may require additional accesses.

□

Proof of step 3. Let $q = \{q_1, \ldots, q_m\}$ be an admissible query union over $\mathcal{C}$. Since $\delta(q_i) \leq \gamma(q_i)$ for every i, we have that $\delta(q) = \max_{i=1,\ldots,m} \delta(q_i) \leq \max_{i=1,\ldots,m} \gamma(q_i)$. Obviously $\max_{i=1,\ldots,m} \gamma(q_i) = \gamma(\bar{q})$ where $\bar{q}$ is the query in q with the maximum cost-depth. But then $\gamma(q)$ is obviously not smaller than $\gamma(\bar{q})$ because it will require all the accesses required for $\bar{q}$ plus possibly some extra accesses required by some other q_i. It follows that

$$\delta(q) = \max_{i=1,\ldots,m} \delta(q_i) \leq \max_{i=1,\ldots,m} \gamma(q_i) = \gamma(\bar{q}) \leq \gamma(q)$$

□

Proof of step 4. Let the admissible query q_j be part of a nr-datalog$^\neg$ program over $\mathcal{C}$. If q_j's body does not contain idb atoms we already know that $\delta(q_j) \leq \gamma(q_j)$. Let us then consider idb atoms of the form $\neg q_i(x_i)$, $i < j$ such that q_i's body does not contain any idb atom. In the graph-depth computation such an atom is accounted for by including $\delta(q_i)$ in a $\max$ operation, therefore $\delta(q_i) \leq \delta(q_j)$ (5.2.1). In the cost-depth computation we have that $\gamma(q_i) \leq \gamma(q_j)$ since all the accesses used in q_i are accounted for in q_j. Since negated idb atoms do not affect hops or accesses in any other way, this situation is analogous to the one for UNICQs at *step 3*, therefore $\delta(q_j) \leq \gamma(q_j)$ if q_j does not contain positive idb atoms. By iterating the same reasoning we can prove that $\delta(q_j) \leq \gamma(q_j)$ holds even if, for each atom $\neg q_i(x_i)$ in q_j's body, q_i contains negated idb atoms.

If instead we consider positive idb atoms of the form $q_i(x_i)$ over the query $q_i(x) \leftarrow conj_i(x, \vec{y})$, $i < j$, appearing in q_j's body, we can reduce q_j to a query not containing any positive idb predicate by iteratively replacing every atom of the form $q_i(x_i)$ with $conj_i(x_i, \vec{y'})$, where $\vec{y'}$ is a new set of variables, until no positive idb predicate appears in q_j's body. We have thus obtained an alternative formulation of q_j which only uses edb, non positive idb and inequality predicates. For this type of query we have already proven that $\delta(q_j) \leq \gamma(q_j)$.

□

6.5.2 Advantages and disadvantages of the two approaches

Having established a relation between graph-depth and cost-depth, we now discuss the practical differences of the two approaches. We start by pointing out that, because of the previous theorem, there are admissible queries where the graph-depth is lower than the cost-depth. This implies that the queries with graph depth not greater than a certain $\bar{k}$

are a subset of the queries that satisfy the same constraint on the cost-depth. We can therefore say that the cost-depth approach is more limiting, since it labels a larger set of queries as "not compliant". The core reason for this difference is, as pointed out earlier in this chapter, that in the graph-depth approach the identifiers who *called* some RAS and the ones who *were called by* some RAS both are at distance 1 from the RAS identifiers, and since the graph-depth is computed as a maximum over all distances, its value will be 1, if no other types of identifiers are involved. On the contrary, in the cost-depth approach the operations of extracting the receivers given the senders and extracting the senders given the receivers are two distinct accesses which, if both are present in a search plan, imply a cost-depth not smaller than 2.

We also point out that the graph-depth approach allows to verify that the guidelines in [2] hold for a query, while the cost-depth approach can lead to rejecting compliant queries. Anyway we point out that the cost-depth approach is more natural for relational databases since it distinguishes different joins between tables. As a final remark, while to compute the graph-depth of a query can be traced back to the shortest path problem, solvable in polynomial time, to compute the cost-depth of a query is NP-hard.

6.6 Conclusive example

We close the chapter by showing a sample SQL query on database *CALLS* for which our approach returns a cost-depth smaller than what the analyst could expect.

Suppose we want to find all the triplets of identifiers, say Alice, Bob and Carl, such that Alice and Bob have been both called by some RAS identifiers, and Carl has been called by both Alice and Bob. The occurrence of such a pattern may raise suspicions about both Alice, Bob and Carl. Here is a simple query achieving this goal:

```
SELECT R1.receiver AS A, R2.receiver AS B, R4.receiver AS C
FROM RAS AS S1, Phone_Call AS R1, Phone_Call AS R2, Phone_Call AS R3,
     Phone_Call AS R4, RAS AS S2
WHERE S1.id = R1.sender AND R1.receiver = R2.sender AND R2.receiver = R3.receiver AND
      R3.sender = R4.receiver AND R4.sender = S2.id AND R1.receiver <> R4.receiver
```

Listing 6.1: Sample query

The corresponding NICQ is

$$q(y,z,u) \leftarrow S(x), R_1(x,y), R_2(y,z), R_3(u,z), R_4(v,u), S(v), y \neq u$$

The query is already minimal. The best search plan for q is

P:
1. $\hat{R}_1(x,y) \leftarrow R_1(x,y), S(x)$
2. $\rho_1(y) \leftarrow \hat{R}_1(x,y)$
3. $\hat{R}_2(y,z) \leftarrow R_2(y,z), \rho_1(y)$
4. $\hat{R}_4(v,u) \leftarrow R_4(v,u), S(v)$
5. $\rho_4(u) \leftarrow \hat{R}_4(v,u)$

6. $\hat{R}_3(u, z) \leftarrow R_3(u, z), \rho_4(u)$
7. $\hat{q}(y, z, u) \leftarrow S(x), \hat{R}_1(x, y), \hat{R}_2(y, z), \hat{R}_3(u, z), \hat{R}_4(v, u), S(v), y \neq u$

The accesses $\hat{R}_1$ and $\hat{R}_4$ are equivalent and so are the extractions ρ_1 and ρ_4. From this it follows that the accesses $\hat{R}_2$ and $\hat{R}_3$ are also equivalent. Since there are two distinct accesses in P, $\gamma(q) = c(P) = 2$.

Algorithm 5: *BranchAndBoundNegated*

Input: A list of accesses $Accesses[q_j]$, a set of negated R atoms $Available$ and an integer c

Output: none

if $Available = \emptyset$ **then**
  $c^* \leftarrow c$
  $Accesses^*[q_j] \leftarrow Accesses[q_j]$
  return
for $\neg R_i(x_i, y_i)$ *in* $Available$ **do**
  for *accesses of the form* $I(x_j) \rightarrow R(x_j, y_j)$ *in* $Accesses[q_j]$ **do**
    if $q_{x_i} \subseteq I$ **then**
      `BranchAndBoundNegated` $(Accesses[q_j], Available \setminus \{\neg R_i(x_i, y_i)\}, c)$
  end
  for *accesses of the form* $I(y_j) \rightarrow R(x_j, y_j)$ *in* $Accesses[q_j]$ **do**
    if $q_{y_i} \subseteq I$ **then**
      `BranchAndBoundNegated` $(Accesses[q_j], Available \setminus \{\neg R_i(x_i, y_i)\}, c)$
  end
  for *each positive atom* A *joined with* $\neg R_i(x_i, y_i)$ *in* q_j*'s body* **do**
    if *the joined variable* z *is not used to access* A **then**
      I **init** $\begin{cases} A & A \text{ is not an } R \text{ atom} \\ \text{the source coming from the access to } A & \text{otherwise} \end{cases}$
      $c' \leftarrow |Accesses[q_j] \cup \{I(z) \rightarrow R(x_i, y_i)\}|$
      if $c' < c^*$ **then**
        `BranchAndBoundNegated`
        $(Accesses[q_j] \cup \{I(z) \rightarrow R(x_i, y_i)\}, Available \setminus \{\neg R_i(x_i, y_i)\}, c')$
  end
end
return

Algorithm 6: *GreedyPlanFinder*

Input: An admissible query q_j over $\mathcal{C}$
Output: An integer Γ representing an upper bound for $\gamma(q_j)$

AG: the search graph of q_j
$Accesses[q_j]$: an empty set of accesses
$Available$: an empty list of arcs
`addAll` $(Available, \{(S_j, R_k, z_i) \mid (S_j, R_k, z_i) \in AG\} \cup \{(q_j, R_k, z_i) \mid (q_j, R_k, z_i) \in AG\})$
for *every idb atom* $(\neg)q_i(x_i)$ *in* q_j*'s body* **do**
    $Accesses[q_j] \leftarrow Accesses[q_j] \cup Accesses[q_i]$
end
$\Gamma \leftarrow |Accesses[q_j]|$
while $Available \neq \emptyset$ **do**
    $c^* \leftarrow \infty$
    `shuffle` $(Available)$
    for $arc \in Available$ **do**
        $c \leftarrow |Accesses[q_j] \cup \{$the access encoded by $arc\}| - |Accesses[q_j]|$
        if $c < c^*$ **then**
            $c^* \leftarrow c$
            $arc^* \leftarrow arc$
    end
    $arc \leftarrow arc^* = (i, j, \ell)$
    $AG \leftarrow$ `PruneGraph`(AG, arc)
    $Accesses[q_j] \leftarrow Accesses[q_j] \cup \{$the access encoded by $arc\}$
    $\Gamma \leftarrow \Gamma + c^*$
    `addAll` $(Available, \{(i', j', \ell') \mid (i', j', \ell') \in Out(j) \wedge j' \notin Chosen\})$
    `removeAll` $(Available, \{(i', j', \ell') \mid (i', j', \ell') \notin AG\})$
end
for *atom* $\neg R(x_i, y_i)$ *in* q*'s body* **do**
    if *there is* ***no*** *access* $I(x_i) \rightarrow R(x_i, y_k)$ *or* $I(y_i) \rightarrow R(x_k, y_i)$ *encoded in* AG **then**
        define a source $I(z_i)$ from an appropriate atom, $z_i \in \{x_i, y_i\}$
        $Accesses[q_j] \leftarrow Accesses[q_j] \cup \{I(z_i) \rightarrow R(x_i, y_i)\}$
        $\Gamma \leftarrow \Gamma + 1$
end
return Γ

CHAPTER 7

IMPLEMENTATION

In this chapter we present a *Java* implementation of the *QueryAnalyzer* system, which, given an input query, is able to compute both its graph-depth and cost-depth. The implemented algorithms perform computations on digraphs and arc-labeled multidigraphs. For this purpose we have used the *JGraphT* library [18] which provides data structures and algorithms which suited our purposes well. We first present the user interface, we then detail the various modules of the program.

Implemented features

The current iteration of *QueryAnalyzer* supports full NICQ queries. The program allows for both graph-depth and cost-depth computation, including query minimization.

7.1 User interface

QueryAnalyzer provides a simple command line interface to the user. An analyst willing to utilize this program just needs to type the query the must be analyzed and which approach to use between graph-depth and cost-depth. If the analyst chooses to compute the graph-depth of the query, algorithm *ComputeGraphDepth* is used. In the case of cost-depth, the user must also specify whether he/she wants to retrieve the exact cost-depth of the query or if it suffices to get a (faster to obtain) upper bound. In the first case, algorithm *BranchAndBoundPlanFinder* is adopted while, in the latter, the computation is carried over by *GreedyPlanFinder*. Once the query is typed, QueryAnalyzer performs the required computation and prints the results. The program is able to understand queries written in *Datalog* notation, e.g. the NICQ

$$q(x, y, z) \leftarrow S_0(x), R_1(x, y), R_2(z, x), S_2(z)$$

is written as `q(x,y,z):-S0(x),R(x,y),R(z,x),S2(z).`

We now provide some examples of the use of QueryAnalyzer: when first running the program the following message is displayed

```
Welcome to QueryAnalyzer, please follow the instructions below.
To compute the graph-depth of a query, type the query followed by ';g',
to compute the exact cost-depth of a query, type the query followed by';p',
to compute an upper bound to the cost-depth of a query, type the query
followed by ';f',
the above commands can be combined as ';gp' or ';gf'.
```

If command “`q(x,y,z) :- S0(x), R(x,y), R(y,z) ;g`” is typed, the following is displayed:

```
The graph-depth is 2
The distance of every variable is:
{z=2, y=1}
```

Along with the graph-depth of the query, the distance of all the considered variables to RAS is displayed. If instead we type command “`q(x,y,z) :- S0(x), R(x,y), R(y,z) ;p`” the following is displayed

```
The cost-depth is 2
The best query plan is [S0 x R1(x,y), R1(x,y) y R2(y,z)]
```

The best query plan, represented by the edges of corresponding access graph, is displayed along with the cost-depth of the query.
Finally, if we type command “`q(x,y,z) :- S0(x), R(x,y), R(y,z) ;f`” we obtain the following:

```
The cost-depth is AT MOST 2
The best query plan is [S0 x R1(x,y), R1(x,y) y R2(y,z)]
```

7.2 Graph-depth module

The graph-module of QueryAnalyzer is represented by the Java class `GraphDepthFinder`. Below is a skeleton of the class[1]

```
1 public class GraphDepthFinder {
2   private int graphDepth = 0;
3   private Map<Variable, Integer> distances = new HashMap<Variable, Integer>();
4   //! getter methods...
5   private void reset() {...}
6   private static SimpleDirectedGraph<Variable, DefaultEdge> buildCallGraph(Query query)
      {...}
7   public void computeGraphDepth(Query query) {...}
8   public void printResults() {...}
9 }
```

Listing 7.1: Class `GraphDepthFinder`

Attribute `graphDepth` represents the graph-depth of the last analyzed query and `distances` contains the distance between any considered variable and the RAS identifiers. Method `reset` simply resets the above variables. Method `buildCallGraph` returns a `SimpleDirectedGraph<Variable, DefaultEdge>` representing the call graph for `query`. Method `printResults` displays the value of the variables above. The most important method of `GraphDepthFinder` is `computeGraphDepth`, whose code is shown in Listing 7.2.

[1] When showing code listings, we will often write comments starting with '!' in place of portions of code which are not relevant to the current discussion.

```
public void computeGraphDepth(Query query) {
    reset();
    query = NICQOptimizer.minimize(query);
    SimpleDirectedGraph<Variable, DefaultEdge> callGraph = buildCallGraph(query);
    Queue<Variable> u = new LinkedList<Variable>();
    for(Variable v : callGraph.vertexSet()) {
      if(v.isBad()) {
        distances.put(v, 0);
        u.add(v);
      } else {
        distances.put(v, Integer.MAX_VALUE);
      } }
    while(!u.isEmpty()) {
      Variable n = u.poll();
      for(DefaultEdge e : callGraph.edgesOf(n)) {
        Variable m = (n.equals(callGraph.getEdgeSource(e)) ? callGraph.getEdgeTarget(e) :
    callGraph.getEdgeSource(e));
        if(distances.get(m) > distances.get(n) + 1) {
          distances.put(m, distances.get(n) + 1);
          u.add(m);
        }
        callGraph.removeEdge(e);
        graphDepth  = Math.max(graphDepth, distances.get(m));
  } } }
```

Listing 7.2: Method `ComputeGraphDepth`

The method is a straightforward implementation of the algorithm presented in Chapter 5.

7.3 Cost-depth module

The core of the cost-depth module is made of classes `BranchAndBoundPlanFinder` and `GreedyPlanFinder`. Both classes implement the abstract class `QueryPlanFinder` below:

```
public abstract class QueryPlanFinder {
  private AccessGraph<AccessNode, AccessEdge> bestGraph = null;
  private Map<AccessNode, String> bestPlan = null;
  private int bestPlanCost = Integer.MAX_VALUE;
  public void reset() {...}
  //! getters and setters...
  public abstract void findBestQueryPlan(String query);
  public abstract void printResults();
}
```

Listing 7.3: Class `QueryPlanFinder`

Field `bestGraph` stores the access graph of the current query, `bestPlan` maps each R atom (positive or negated) to a `String` representation of the best access for it and `bestPlanCost` stores the cost of such plan. Method `reset` resets the above fields and is called at the start of the analysis of each query. Method `findBestQueryPlan` literally finds the best

canonical query plan for `query`. Method `printResults` is used to display `bestPlan` and `bestPlanCost` after the computation ends. The `AccessGraph` type used in this module is a subclass of the `DirectedMultigraph` type provided by JGraphT. Types `AccessNode` and `AccessEdge` represent the nodes and the arcs of an access graph respectively.

We now take a look at the two different implementations of method `findBestQueryPlan`.

7.3.1 *BranchAndBoundPlanFinder*

Class `BranchAndBoundPlanFinder` implements the algorithm of the same name presented in Chapter 6. Note that arcs are referred to as *edges* in this context. This class implements method `findBestQueryPlan` in the following way:

```
1  public void findBestQueryPlan(String query) {
2      reset();
3      query = new Query(input);
4      query = NICQOptimizer.minimize(query);
5  AccessGraph<AccessNode, AccessEdge> graph = buildAccessGraph(query.toString());
6      List<AccessEdge> available = new LinkedList<AccessEdge>();
7      Map<AccessNode, String> plan = new HashMap<AccessNode, String>();
8      for(AccessEdge edge : graph.edgeSet()) {
9        if(edge.getSourceDesc().startsWith("S")) {
10         available.add(edge);
11       }
12     }
13     planSize = 0;
14     for(AccessNode n : graph.vertexSet()) {
15       if(n.getType()==RelationType.R) {
16         planSize++;
17       }
18     }
19     recursiveQueryPlanFinder(graph, available, plan, 0);
20   }
```

Listing 7.4: Method `findBestQueryPlan` as implemented in `BranchAndBoundPlanFinder`

This method corresponds to algorithm *BranchAndBoundPlanFinder* in Chapter 6: it minimizes the input query, build its access graph, initializes the data structures `available` and `plan` and calls method `recursiveQueryPlanFinder` which corresponds to algorithm `BranchAndBoundRecursive` in Chapter 6. The code for `recursiveQueryPlanFinder` is now shown. We first show the base case of the algorithm and then provide the code for the key blocks dubbed “choosing edge” and “discarding edge”.

```
private void recursiveQueryPlanFinder(AccessGraph<AccessNode, AccessEdge> graph,
    List<AccessEdge> available, Set<AccessNode> chosen, int cost) {
    if(available.isEmpty()) {
      if(chosen.size() == planSize) {
        setBestGraph(graph);
        Set<Atom> negatedAtoms = new HashSet<Atom>();
        for(Atom a : query.getBody()) {
          if(a.isR() && a.isNegated()) {
            negatedAtoms.add(a);
          }
        }
        branchAndBoundNegated(negatedAtoms, plan, cost);
      }
      return;
    }
    AccessEdge edge = available.remove(0);
    AccessNode accessed = getNode(edge.getTargetDesc());

    //! choosing edge

    //! discarding edge
  }
```

Listing 7.5: Method `recursiveQueryPlanFinder`

We point out that, once no more edges are available, `branchAndBoundNegated` is called to handle the negated R predicates.

```
AccessEdge edge = available.remove(0);
AccessNode accessed = getNode(edge.getTargetDesc());
// choosing edge
Map<AccessNode, String> plan2 = new HashMap<AccessNode, String>(plan);
List<AccessEdge> available2 = new LinkedList<AccessEdge>(available);
int cost2 = 0;
String sourceDesc = edge.getSourceDesc();
AccessNode source = getNode(sourceDesc);
String accessVar = edge.getLabel();
if(source.getType() == RelationType.S) {
  if(accessVar.equals(accessed.getSender())) {
    plan2.put(accessed, sourceDesc + "$s");
  } else {
    plan2.put(accessed, sourceDesc + "$r");
  }
} else if(source.getType() == RelationType.R) {
  if(accessVar.equals(accessed.getSender())) {
    plan2.put(accessed, plan2.get(source) + "s");
  } else {
    plan2.put(accessed, plan2.get(source) + "r");
  }
}
AccessGraph<AccessNode, AccessEdge> graph2 = pruneGraph(graph, edge);
cost2 = cost + (plan.values().contains(plan2.get(accessed)) ? 0 : 1);
if(cost2 < getBestPlanCost()) {
  for (AccessEdge outgoing : graph2.outgoingEdgesOf(accessed)) {
    if(!plan2.keySet().contains(getNode(outgoing.getTargetDesc()))) {
      // all edges starting from accessed and ending into non-accessed nodes are taken
      available2.add(outgoing);
    }
  }
  Iterator<AccessEdge> it = available2.iterator();
  while(it.hasNext()) {
    AccessEdge e = it.next();
    if(!graph2.edgeSet().contains(e)) {
      // all pruned edges in the graph are removed from available2
      it.remove();
    }
  }
  recursiveQueryPlanFinder(graph2, available2, plan2, cost2);
}
```

Listing 7.6: The “choosing edge” part of method `recursiveQueryPlanFinder`

```
if(graph.incomingEdgesOf(accessed).size() > 1) { // still other options to access 'accessed'
      // discarding edge
      Map<AccessNode, String> plan1 = new HashMap<AccessNode, String>(plan);
      List<AccessEdge> available1 = new LinkedList<AccessEdge>(available);
      recursiveQueryPlanFinder(pruneGraph(graph, null), available1, plan1, cost);
    }
```

Listing 7.7: The "discarding edge" part of method `recursiveQueryPlanFinder`

Exactly as in the *BranchAndBoundPlanFinder* algorithm, in Listing 7.7, the new arc `edge` is discarded. In Listing 7.6 the new arc is instead added to the solution and its cost is computed, by checking if the access, represented by a `String` is contained in the accesses made so far (`plan`). If the new cost is lower than the cost of the best solution so far, then `available` is updated by adding all the arcs that can be now chosen because of the addition of `edge` and the method is recursively called.

7.3.2 *GreedyPlanFinder*

Class `GreedyPlanFinder` implements the algorithm of the same name presented in Chapter 6. Method `findBestQueryPlan` is implemented as follows:

```
public void findBestQueryPlan(String query) {
    AccessGraph<AccessNode, AccessEdge> graph = buildAccessGraph(query);
    reset();
    Query query = new Query(input);
    AccessGraph<AccessNode, AccessEdge> graph = buildAccessGraph(input);
    List<AccessEdge> available = new LinkedList<AccessEdge>();
    Map<AccessNode, String> plan = new HashMap<AccessNode, String>();
    for(AccessEdge edge : graph.edgeSet()) {
      if(edge.getSourceDesc().startsWith("S")) {
        available.add(edge);
      }
    }
    for(AccessNode n : graph.vertexSet()) {
      if(n.getType()==RelationType.R) {
      }
    }
    fastPlanFinder(query, graph, available, plan);
  }
```

Listing 7.8: Method `findBestQueryPlan` as implemented in `GreedyPlanFinder`

This method sets up `graph`, `available` and `plan` very similarly to what done by `BranchAndBoundPlanFinder`. It then calls method `fastPlanFinder`, shown below.

```
private void fastPlanFinder(AccessGraph<AccessNode, AccessEdge> graph,
      List<AccessEdge> available, Set<AccessNode> chosen) {
      setBestPlanCost(0);
      while(!available.isEmpty()) {
        int bestCost = Integer.MAX_VALUE;
        //! bestEdge, bestGraph and bestChosen are defined
        Collections.shuffle(available);
        for(int i = 0; i < available.size(); i++) {
          //! the edge which seems the best is found
        }
        graph = bestGraph;
        chosen = bestChosen;
        setBestPlanCost(getBestPlanCost() + bestCost);
        available.remove(bestEdge);
        //! available is updated as in BranchAndBoundPlanFinder
      }
      //! here we handle the negate R atoms
      if(getBestPlanCost() > 0) {
        setBestPlan(graph);
      } else {
        reset();
      }
  }
```

Listing 7.9: Method `fastPlanFinder`

The method basically keeps selecting the edge with smaller cost in `available` until `available` is empty. After the best edge is found, `available` is updated exactly as in method `recursiveQueryPlanFinder` of class `BranchAndBoundPlanFinder`. As pointed out in Chapter 6, we call `Collections.shuffle` on `available` since the outcome of the computation depends on the order of the edges in `available`. To find out the best edge, we evaluate its cost as done in `BranchAndBoundPlanFinder`. The `newEdge` with smallest cost becomes `bestEdge`. We now show a snippet from the "negated" section of the algorithm, specifically where we check for reusable accesses in the plan.

```
for(Atom nR : query.getBody()) {
        if(nR.isR() && nR.isNegated()) {
          String sender = nR.getVars().get(0).getVar();
          String receiver = nR.getVars().get(1).getVar();
          boolean found = false;
          for(AccessEdge e : graph.edgeSet()) {
            AccessNode R = graph.getEdgeTarget(e);
            if(sender.equals(R.getSender()) && R.getSender().equals(e.getLabel())
                || receiver.equals(R.getReceiver()) && R.getReceiver().equals(e.getLabel()))
     {
              plan.put(new AccessNode(i--, sender, receiver), plan.get(R));
              found = true;
              break;
            }
          }
          // ! we add an extra access for nR
        }
      }
```

Listing 7.10: Dealing with negated R atoms in method `fastPlanFinder`

7.4 Main module

The main module is made of the Java class `QueryAnalyzer`. This class provides the user interface, and calls the graph-depth and cost-depth modules according to the user's requests. We now show the code for the class.

```
1  public class QueryAnalyzer {
2    //! auxiliary variables
3    private GraphDepthFinder graph = null;
4    private BranchAndBoundPlanFinder bnb = null;
5    private GreedyPlanFinder gree = null;
6    public void computeGraphDepth(String query) {...}
7    public void computeCostDepth(String query, Mode mode) {...}
8    private void initializeGDFinder() {
9      graph = (graph == null ? new GraphDepthFinder() : graph);
10   }
11   private void initializeBnBFinder() {
12     bnb = (bnb == null ? new BranchAndBoundPlanFinder() : bnb);
13   }
14   private void initializeGrFinder() {
15     gree = (gree == null ? new GreedyPlanFinder() : gree);
16   }
17   public static void main(String [] args) {
18     //! provides the user interface
19     //! calls computeGraphDepth and computeCostDepth depending on the user's requests
20   }
21 }
```

Listing 7.11: Class `QueryAnalyzer`

The class has a private attribute for each algorithmic class (`GraphDepthFinder`, `BranchAndBoundPlanFinder`, `GreedyPlanFinder`). These attributes are used in methods `computeGraphDepth` (Listing 7.13) and `computeCostDepth` (Listing 7.12). The `main` method displays the text interface, reads the user commands and calls `computeGraphDepth` and `computeCostDepth` accordingly.

```
public void computeCostDepth(String query, Mode mode) {
    switch(mode) {
    case PRECISE:
      initializeBnBFinder();
      bnb.findBestQueryPlan(query);
      bnb.printResults();
      break;
    case FAST:
      initializeGrFinder();
      gree.findBestQueryPlan(query);
      gree.printResults();
      break;
    default:
      break;
    }
  }
```

Listing 7.12: Method `computeCostDepth` of class `QueryAnalyzer`

```
public void computeGraphDepth(String query) {
    initializeGDFinder();
    graph.computeGraphDepth(query);
    graph.printResults();
  }
```

Listing 7.13: Method `computeGraphDepth` of class `QueryAnalyzer`

CHAPTER 8

EXPERIMENTAL RESULTS

In this chapter we will compare the performance of the three algorithms, as implemented in *QueryAnalyzer*, on a number of randomly generated queries. The aim of these experiments is both to quantify the running time of each algorithm and to measure the quality of the upper bound provided by `GreedyPlanFinder` with respect to the cost-depth computed by `BranchAndBoundPlanFinder`.

8.1 Conducted experiments

We have run the two algorithms on three sets of randomly generated admissible queries over $\mathcal{C}$. Each set of queries was generated by specifying values for the following four parameters: the number of variables in the query (v), the number of atoms over R in the query (r), the number of negated R atoms in the query (nr), the number of inequalities in the query (neq), the number of used seeds in the query (s) and the probability for any variable to represent a seed identifier (p). For v we have chosen all values from 2 to 10, because we have assumed that rarely the analyst will input queries with more than 10 variables, especially considering that, in [2], the maximum number of hops is stated to be 3 and to reach such distance it is sufficient to have 4 variables. For what concerns r we have experimented with three values: $v-1$, $\lceil \frac{3}{2}(v-1) \rceil$ and $2(v-1)$. The first value is the minimum required to reach the maximum number of hops using v variables while the third value is the double of the first one which we assume will often suffice for the analyst's needs ($r = 18$ when $v = 10$). Finally the second value is the (rounded) mid-point between the first and the third. We have assumed the number of negated R atoms and inequalities to be equal to $\lceil \frac{r}{3} \rceil$ and $\lceil \frac{r}{4} \rceil$ respectively. We have conducted the experiments with a number $s = 1$ and $s = 3$ of seeds because we think that the most common queries retrieve identifiers related to just a few specific seeds. Finally, we have parameter p which represents, for every variable x_j, the probability of some atom $S_k(s_j)$ to appear in the query. We have set $p = 0.25$ for every query. This is similar to generate queries in which one fourth of the variables represent suspects, but we have chosen to specify a probability rather than an absolute number to obtain a richer set of queries. Needless to say, all the generated queries were admissible according to definition 5.1.2. The three sets of generated queries are the following:

Trial 1. 100 queries for each $v = 2, \ldots, 10$, with $r = v - 1$, $nr = \lceil \frac{r}{3} \rceil$, $neq = \lceil \frac{r}{4} \rceil$, $s = 1$ and $p = 0.25$.

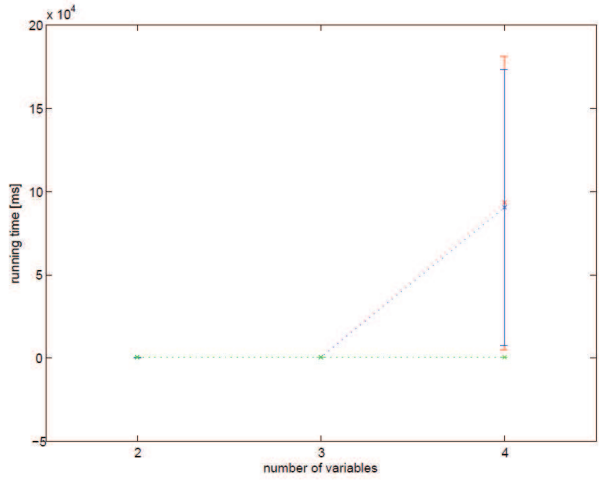
Trial 2. 100 queries for each $v = 2, \ldots, 10$, with $r = \lceil \frac{3}{2}(v-1) \rceil$, $nr = \lceil \frac{r}{3} \rceil$, $neq = \lceil \frac{r}{4} \rceil$, $s = 3$ and $p = 0.25$.

Trial 3. 30 queries for each $v = 2, \ldots, 10$, with $r = 2(v-1)$, $nr = \lceil \frac{r}{3} \rceil$, $neq = \lceil \frac{r}{4} \rceil$, $s = 3$ and $p = 0.25$.

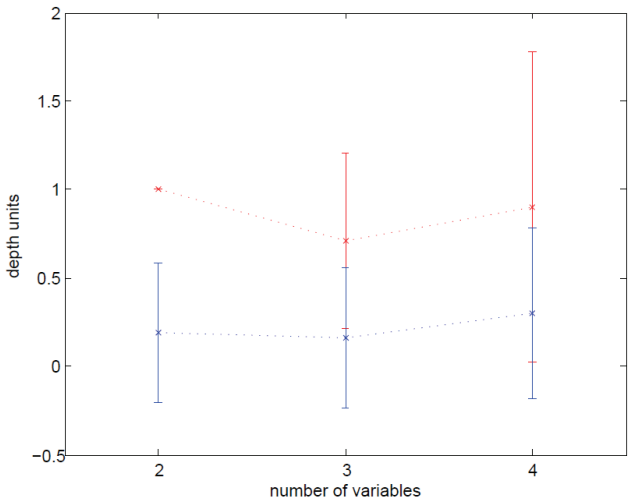
For each query in each set, we have analyzed the query using both `GraphDepthFinder`, `BranchAndBoundPlanFinder` and `GreedyPlanFinder`. All the experiments have been conducted on a 2.40GHz quad-core Intel-based machine with 8GB of RAM.

8.2 The cost of minimization

As we have pointed out many times, NICQ minimization is Π_2^p-complete. We have implemented it as described in [11] and run it inside both `GraphDepthFinder` and `BranchAndBoundPlanFinder`. The results obtained on *Trial 1* are shown in Image 8.2



The red error plot represents the average and standard deviation of `GraphDepthFinder`'s running time. The blue and green points achieve the same for `BranchAndBoundPlanFinder` and `GreedyPlanFinder` respectively. We do not show results for $v > 4$ because the minimization algorithm has made it impossible for the whole procedure to complete. The behavior depicted by the graph is coherent with out prior knowledge: the greedy algorithm (GR), free from the burden of minimization, runs much faster than the other two. Of course the graph depth algorithm (GD) is much faster than the branch & bound (BB) after minimization, but their overall performance is comparable because of the shared complexity of minimization. Let us now take a look at the difference between the cost-depth computed by BB and the upper bound computed by GR.



The blue error plot shows average and standard deviation of the difference between GR's approximation and the real cost-depth provided by BB. The red error plot shows the difference between the cost-depth and the graph-depth. We can see that, even with the minimization at work, GR manages to follow BB almost precisely, and with no notable difficulties as the number of variables increases

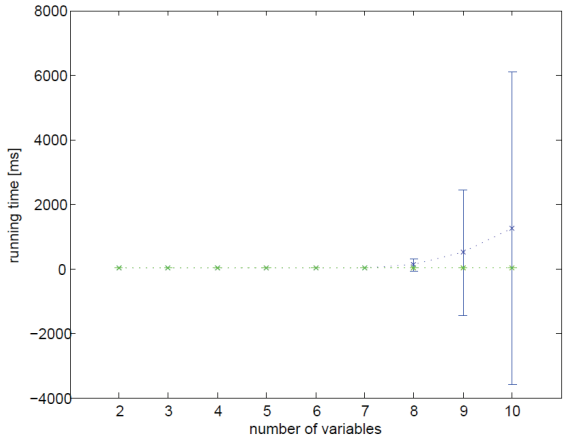
The above results show that performing minimization is impractical as query get more complex, therefore the query analyst will have to rely on his/her ability to write minimal queries. For this reason, all the following trials have been conducted *excluding* minimization on both GD and BB.

8.3 Time performances

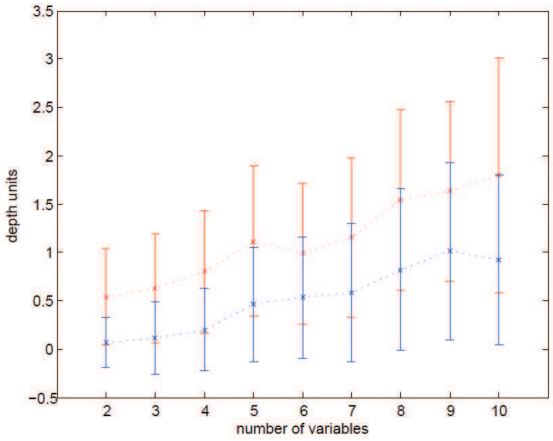
For each of the three trials above, we have measured the running time of each algorithm and computed average (in seconds) and standard deviation (in nanoseconds) for each value of v. We now present and discuss the obtained results.

8.3.1 Trial 1

For the first set of queries we have obtained the results in Image 8.3.1.

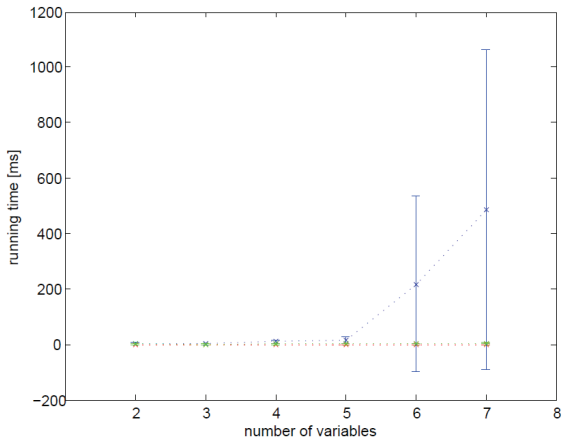


In Image **??** we can find the depth approximations as in the previous paragraph.

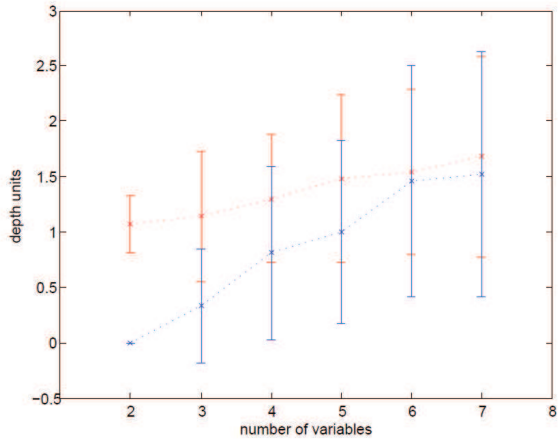


8.3.2 Trial 2

For the second set of queries we have obtained the results in Image 8.3.2.

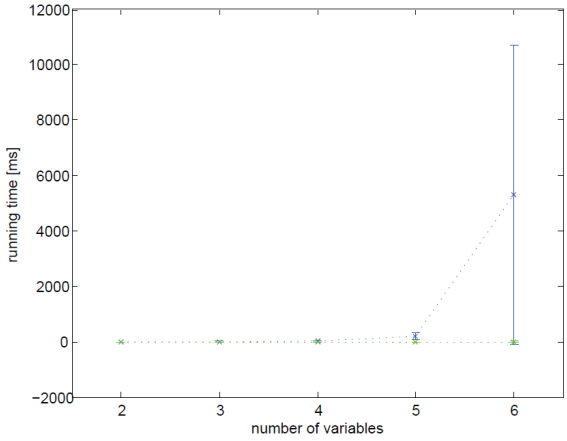


In Image **??** we can find the depth approximations.

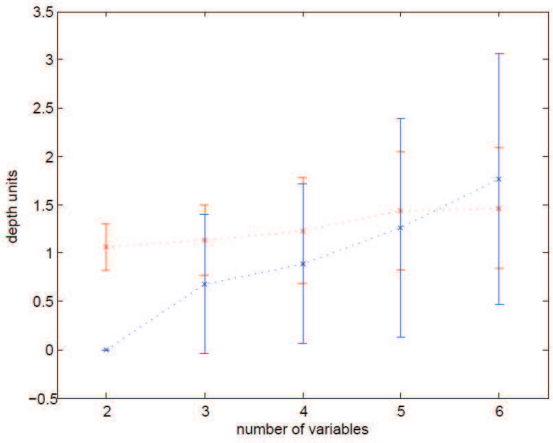


8.3.3 Trial 3

Last, for the third set of queries we have obtained the results in Image 8.3.3.



In Image **??** we can find the depth approximations.



8.4 Final observations

Summing up the results of the above trials, we can say that `BranchAndBoundPlanFinder` cannot be effectively used for queries with large values of v and r, forcing the analyst to employ approximate algorithms, such as `GreedyPlanFinder`. Thankfully, it turns out that this second algorithm presents acceptable running times even for the largest queries we have generated and does provide an upper bound to the cost-depth of the query that is generally much closer to the value computed by B&B than the trivial upper bound r.

The obtained results suggest an efficient strategy for the analyst: suppose that the guidelines imposed on the executable queries state that the cost-depth of a query cannot be greater than k. To check the compliance of a query q to such requirement, the analyst

first runs `GreedyPlanFinder` which returns value k_g. If $k_g \leq k$, the analyst can execute q without any problems, otherwise, he/she will run `BranchAndBoundPlanFinder` to obtain some $k_b \leq k_g$. The results obtained by the graph depth algorithm are instead acceptable.

CHAPTER 9

CONCLUSIONS

9.1 Obtained results

In this thesis we have proposed two approaches to ensure the legality of queries on telephony metadata. The advantages of both approaches with respect to the state of the art are that the metrics they provide do not depend on the content of the database but solely on the query and in order to compute such metrics, the query does not need to be executed. Based on the guidelines described in [1, 2], we have proposed a *graph-depth* approach that computes the maximum number of *hops* from the RAS (Reasonable Articulable Suspicion) identifiers at which any identifier returned by a query lies. The approach is based on an algorithm whose running time grows as a polynomial of the size of the input query.

We have also proposed a cost-depth approach, based on the concept of access limitations to databases, that computes the minimum number of *accesses* that must be performed to answer the query. Since to compute the cost-depth of a query is NP-hard, we have proposed two algorithms, a *Branch & Bound* (B&B) algorithm which returns exactly the above number and a *Greedy* algorithm, which returns an upper bound to the above number. We have proved that the cost-depth approach is more restrictive than the graph-depth approach, therefore the choice of which one to use depends on the guidelines that the intelligence agency has to comply with.

We have implemented all the above algorithms as part of the *QueryAnalyzer* system, written in Java, which allows an intelligence analyst to input a query and display the metrics computed by the above algorithms. This tool can be used by the analyst before executing a query, to make sure that all the returned identifiers are legally obtainable.

Finally, we have run the algorithms on a number of randomly generated queries and we have found that, while the B&B algorithm provides a more precise result, its running time grows very quickly with the complexity of the query. Luckily, our experiments showed that the Greedy algorithm is generally much faster and, while it does not compute the exact cost-depth value, the obtained results are usually very close to the ones provided by the B&B algorithm. Therefore, the analyst can initially employ the Greedy algorithm and, if the obtained results are not precise enough to decide if the query complies with the guidelines, the B&B algorithm can be used to determine the compliance of the query.

The system we have developed could be used by intelligence agencies worldwide whose searches on telephony metadata are subject to restrictions similar to the ones in [2]. By inserting *QueryAnalyzer* into the analyst work flow, the agency would guarantee that, if the results of a query considered to be illegal were ever to be used, then the analyst knew it or at least had the tools to check the query for compliance before executing it. Of course

employing *QueryAnalyzer* would require minor modifications depending on the specific requirements of every agency and the format of the queried data. However, we point out that the simplicity of the database schema that we have assumed makes it very easy to apply the approach to analyze any kind of metadata describing the communication between two parties, such as e-mails, instant messaging systems, activity on social networks, etc.

9.2 Further work

The research conducted in this thesis can be expanded. For instance, the cost-depth approach may be expanded so that, if a query q is found to be requiring more accesses than allowed, we can derive an *approximated* query q' such that $q'^D \subset q^D$ and q' does not require more accesses than what is allowed.

Finally, from a more practical point of view, *QueryAnalyzer* may be improved in various ways: the implementation could be improved to handle full NICQ queries and nr-datalog$^\neg$ programs and the textual interface could be replaced by a graphical interface which allows the analyst to design a query by drawing its call graph. This way, the analyst would be able to focus solely on what the query must achieve, avoiding to be distracted by a possibly cumbersome syntax.

CITED LITERATURE

1. FISA Ct: In re Production of Tangible Things From [REDACTED]. Order, Number BR 08-13, March 2009. http://www.fas.org/irp/agency/doj/fisa/fisc-021209.pdf.
2. US Department of Justice: Administration white paper: Bulk collection of telephony metadata under section 215 of the USA Patriot Act. Available from https://www.aclu.org/nsa-documents-released-public-june-2013, August 2013.
3. Kanich, C., Panebianco, A., Sloan, R. H., Warner, R., and Zuck, L. D.: A Modest Proposal to the NSA. Unpublished, 2013.
4. Panebianco, A.: Simple Structures For Complex Data: Optimizing metadata storage & retrieval using graph databases. Master's thesis, University of Illinois at Chicago, 2014.
5. Office of the Director of National Intelligence (2015, February 27): Joint Statement by the Department of Justice and the Office of the Director of National Intelligence on the Declassification of Renewal of Collection Under Section 215 of the USA PATRIOT Act (50 U.S.C. Sec. 1861) [Web blog post]. Retrieved April 10, 2015, from http://icontherecord.tumblr.com/post/112255431548/joint-statement-by-the-department-of-justice-and.
6. solid IT: DB-Engines Ranking - popularity ranking of database management systems. Retrieved April 10, 2015, from http://db-engines.com/en/ranking.
7. Li, C. and Chang, E. Y.: Answering queries with useful bindings. ACM Trans. Database Syst., 26(3):313–343, 2001. http://doi.acm.org/10.1145/502030.502032.
8. Calì, A. and Martinenghi, D.: Querying data under access limitations. In Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, ICDE '08, pages 50–59, Washington, DC, USA, 2008. IEEE Computer Society.
9. Braga, D., Ceri, S., Daniel, F., and Martinenghi, D.: Optimization of multi-domain queries on the web. PVLDB, 1(1):562–573, 2008. http://www.vldb.org/pvldb/1/1453918.pdf.
10. Calì, A. and Calvanese, D.: Optimising query answering in the presence of access limitations (position paper). In Database and Expert Systems Applications, 2006. DEXA '06. 17th International Workshop on, pages 547–552, 2006.
11. Ullman, J. D.: Information integration using logical views. In Proceedings of the 6th International Conference on Database Theory, ICDT '97, pages 19–40, London, UK, UK, 1997. Springer-Verlag.
12. Abiteboul, S., Hull, R., and Vianu, V.: Foundations of Databases. Addison-Wesley, 1995.

13. Li, C. and Chang, E. Y.: Query planning with limited source capabilities. In ICDE, pages 401–412, 2000.
14. Halevy, A. Y.: Answering queries using views: A survey. VLDB J., 10(4):270–294, 2001. http://dx.doi.org/10.1007/s007780100054.
15. Balakrishnan, V. K.: Graph Theory. McGraw-Hill, 1997.
16. Romano, R. (2013, June 18): 3 Billion Phone Calls Made in US Every Day. Retrieved April 2, 2015, from http://www.texasinsider.org/3-billion-phone-calls-made-in-us-every-day/.
17. Bernhard, K. and Jens, V.: Combinatorial Optimization: Theory and Algorithms (5 ed.). Springer, 2012.
18. The JGraphT team: JGraphT - a free Java Graph Library. Retrieved February 14, 2015, from http://jgrapht.org.
19. Calì, A., Calvanese, D., and Martinenghi, D.: Dynamic query optimization under access limitations and dependencies. J. UCS, 15(1):33–62, 2009. http://www.jucs.org/jucs_15_1/dynamic_query_optimization_under.