UMONS
Faculté des Sciences
Département d'Informatique

# Certain Query Answering in First-Order Languages

**Alexandre Decan**

A dissertation submitted in fulfillment of the requirements of the degree of *Docteur en Sciences*

**Advisor**
Prof. Dr. Jef Wijsen
Université de Mons, Belgique

**Jury**
Prof. Dr. Véronique Bruyère
Université de Mons, Belgique
Dr. Olivier Delgrange
Université de Mons, Belgique
Dr. Olivier Gauwin
Université Bordeaux 1, France
Prof. Dr. Floris Geerts
Universiteit Antwerpen, Belgique
Prof. Dr. Tom Mens
Université de Mons, Belgique
Prof. Dr. Slawomir Staworko
Université de Lille 3, France
Prof. Dr. Jef Wijsen
Université de Mons, Belgique

July 2013

# Acknowledgments

First and foremost, I would like to thank my advisor, Jef Wijsen, for giving me the opportunity to experience research, for his guidance and his support. This dissertation would not have been possible without his advice and discussions.

I would like to thank all the collaborators I've worked with, Olivier Gauwin, Olivier Carton, Fabian Pijcke and especially Véronique Bruyère for her precious advice.

I also thank my friends and colleagues for their support and comments. Sylvain, Térence, Mathieu, Stefan, the discussions we had were very valuable to me. I learned a lot of things from you, related or not to our work. Thank you for your company and interest.

Above all, I would like to thank my wife, Perrine. Thank you for your love, your support and for our adorable son, Simon. Thank you both for being there.

# Abstract

Real-life databases often store uncertain, inconsistent or incomplete information. The term *certain query answering* refers to methods for computing reliable answers to queries on such databases. This thesis focuses on cases where this computation is expressible in first-order logic (and hence in the low complexity class $\mathbf{AC^0}$).

The first part of the thesis deals with certain query answering on relational databases that are allowed to violate primary key constraints, also called uncertain databases. A repair of such an uncertain database is obtained by selecting a maximal number of tuples of it without ever selecting two distinct tuples of the same relation that agree on their primary key. We say that a Boolean query $q$ is certain in an uncertain database $\mathsf{db}$ if $q$ evaluates to true on every repair of $\mathsf{db}$. Given a Boolean query $q$, $\mathsf{CERTAINTY}(q)$ is defined as the set of uncertain databases in which $q$ is certain. It is known that there exist conjunctive queries $q$ for which $\mathsf{CERTAINTY}(q)$ is **coNP**-hard. This thesis focuses on the case where $\mathsf{CERTAINTY}(q)$ is first-order definable, meaning that there exists a first-order sentence $\varphi$ to determine whether $q$ is certain. Such sentence $\varphi$ is also called a certain first-order rewriting for $q$. Given an acyclic Boolean self-join-free conjunctive query $q$, it is decidable whether $\mathsf{CERTAINTY}(q)$ is first-order definable. We provide algorithms to construct first-order rewritings for such queries, both in first-order logic and in SQL. We study the effect of syntactic simplifications on the execution time of certain SQL rewritings on large databases.

In the second part of the thesis, motivated by a problem in temporal databases, we consider uncertainty in the framework of first-order logic on words. In this setting, uncertainty is captured by the concept of multiword, which is a finite sequence of nonempty sets of possible symbols. Every word obtained by selecting one symbol from each set is a possible word. Given a word $w$, $\mathsf{CERTAIN}(w)$ is the set of multiwords such that $w$ is a factor of every possible word of the multiword. We provide strong supporting evidence for the conjecture that $\mathsf{CERTAIN}(w)$ is first-order definable for every word $w$. We also study the deterministic finite automata recognizing $\mathsf{CERTAIN}(w)$.

# Résumé

Les bases de données actuelles contiennent bien souvent des informations incertaines, inconsistantes ou incomplètes. Le terme *certain query answering* (CQA) qualifie les méthodes utilisées pour calculer les réponses fiables à des requêtes sur ces bases de données. Cette thèse se concentre sur les cas où un tel calcul est exprimable en logique du premier ordre (et donc, dans la classe de faible complexité $\mathbf{AC^0}$).

La première partie de la thèse traite du CQA dans des bases de données relationnelles qui peuvent ne pas satisfaire des contraintes de clés primaires. Ces bases de données sont également appelées “incertaines”. Une réparation d'une telle base de données incertaine est obtenue en sélectionnant un nombre maximum de tuples sans jamais sélectionner deux tuples distincts d'une même relation, et qui partagent une même valeur de clé primaire. Une requête booléenne $q$ est dite “certaine” dans une base de données incertaine $\mathsf{db}$ si $q$ s'évalue à vrai sur chaque réparation de $\mathsf{db}$. Étant donnée une requête booléenne $q$, $\mathsf{CERTAINTY}(q)$ est défini comme l'ensemble des bases de données incertaines dans lesquelles $q$ est certaine. La littérature fait déjà mention de requêtes conjonctives $q$ pour lesquelles $\mathsf{CERTAINTY}(q)$ est **coNP**-dur. Cette thèse se focalise sur les cas où $\mathsf{CERTAINTY}(q)$ est définissable en logique du premier ordre, c'est-à-dire qu'il existe une phrase $\varphi$ en logique du premier ordre qui détermine si $q$ est certaine. Une telle phrase $\varphi$ est également appelée une ré-écriture certaine de $q$ en logique du premier ordre. Si $q$ est une requête booléenne, acyclique, conjonctive et sans self-join, le problème de savoir si $\mathsf{CERTAINTY}(q)$ est définissable en logique du premier ordre est décidable. Nous proposons des algorithmes qui construisent des ré-écritures certaines pour de telles requêtes, à la fois en logique du premier ordre et en SQL. Nous étudions ensuite les effets qu'ont plusieurs simplifications syntaxiques sur les temps d'exécution des ré-écritures certaines en SQL sur de larges bases de données.

Dans la seconde partie de la thèse, poussés par une problématique dans les bases de données temporelles, nous considérons les incertitudes dans le cadre de la logique du premier ordre sur les mots. Dans ce contexte, l'incertitude est retranscrite au sein du concept de multimot. Un multimot est une séquence finie d'ensembles (non-vides) de symboles possibles. Chaque mot pouvant être obtenu en choisissant un symbole dans chaque ensemble de symboles possibles est un mot possible. Si $w$ est un mot, alors $\mathsf{CERTAIN}(w)$ est défini comme l'ensemble des multimots tels que $w$ est facteur de chaque mot possible du multimot. Nous apportons des preuves solides qui soutiennent la conjecture que $\mathsf{CERTAIN}(w)$ est définissable en logique du premier ordre, quelque soit le mot $w$. Nous étudions également les automates déterministes finis reconnaissant $\mathsf{CERTAIN}(w)$.

# Table of Contents

CHAPTER 1

# Databases and Uncertainty

A database that does not satisfy all of its integrity constraints is an inconsistent database. While inconsistency is generally an undesirable property of a database, this does not mean that an inconsistent database is completely unreliable: tuples involved in some integrity constraint violation may contain both consistent and inconsistent data. In general, there exist many sensible ways to make an inconsistent database consistent. Intuitively, the consistent part of the database contains information that is invariant regardless of the way chosen to restore consistency.

This chapter is organized as follows: Section 1.1 presents the basics of database theory needed in this thesis. We briefly recall the fundamentals of databases and of first-order logic as a query language. Section 1.2 investigates what happens when the database does not satisfy some constraints. Integrity constraints are introduced in Subsection 1.2.1 while Subsection 1.2.2 considers inconsistency by integrity constraint violations. Subsection 1.2.3 gives a characterization of the data that are consistent and states the general problem of consistent query answering: given a query, what is the answer to this query that is consistent? In particular, we focus on uncertain databases: databases that are allowed to violate primary keys. In Subsection 1.2.4, we are interested in computing the answer that is certain by means of a technique known as certain first-order rewriting.

At the end of this chapter, we introduce two interesting problems in the field of consistent query answering and give a general outline of this thesis.

## 1.1 Databases Fundamentals

This section is mainly based on [Abiteboul et al. 1995] and [Maier 1983].

### 1.1.1 The Relational Model

| Emp | Name | WorksFor | Age |
|---|---|---|---|
| | Paul | Microsoft | 60 |
| | Bill | Microsoft | 57 |
| | Larry | Google | 40 |
| | Tim | Apple | 52 |

| Comp | Name | Location |
|---|---|---|
| | Microsoft | Redmond |
| | Google | Mountain View |
| | Apple | Cupertino |

Figure 1.1: Database $\mathsf{db}_1$ of Example 1.1.

The *relational database model* was introduced by Codd in 1969 [Codd 1969]. Although the relational database model is unique, it has a variety of "implementations" and the term is commonly used to refer to a broad class of database models that have relations as data structures and languages to specify queries, updates, and integrity constraints.

Assume two disjoint sets **dom** and **var** of respectively *constants* and *variables*. Constants and variables are *symbols*. Assume that a countably infinite set **attr** of *attributes* is fixed.

**Example 1.1** Figure 1.1 shows a conventional representation of a relational database. Database $\mathsf{db}_1$ contains two tables, the relations Emp and Comp. The column headers in each table are the attributes of the corresponding relations. Rows in each table are tuples. Each value in a row is an element of **dom**, the domain of the database, and is called a constant. ◁

A *tuple* over a (possibly empty) finite set $U = \{u_1, \ldots, u_n\}$ of attributes is a total mapping $t$ that associates to each attribute $x$ in $U$ a value $t(x)$ in **dom**. Commonly, a total order is assumed on the set of attributes. In that case, a tuple $t = \{u_1 : a_1, \ldots, u_n : a_n\}$ can be specified as a finite sequence $\langle a_1, \ldots, a_n \rangle$. If $i$ is a position with $1 \leq i \leq n$, then we use $t \cdot i$ to access the $i$-th element of the tuple, that is, $t \cdot i = t(u_i)$. If $X$ is a subset of $U$, $t[X]$ denotes the tuple $v$ over $X$ such that $v(u) = t(u)$ for each $u \in X$.

A *relational database schema* is a finite set of *relation names*, each with an associated ordered finite set of attributes. Let $R$ be a relation name and $U$ the set of attributes of $R$. We denote by $sort(R)$ the set $U$ of attributes. The arity of $R$, denoted by $\mathsf{arity}(R)$, is the number of attributes in $sort(R)$. Attributes do not need to have names: in this case, each attribute is identified by its position, from 1 to $\mathsf{arity}(R)$. If $R$ is a relation name whose arity is $n$, then $R(a_1, \ldots, a_n)$ is an $R$-atom (or simply an *atom*) where each $a_i$ $(1 \leq i \leq n)$ is a constant in **dom** or a variable in the set **var** of variables. If an atom has no variable, then it is *ground* and it is called a *fact*.

A *relation over $U$* (or simply a relation if $U$ is understood) is a finite set of tuples over $U$. If $\mathbf{S}$ is a database schema, a *database* over $\mathbf{S}$ associates to each relation name $R$ in $\mathbf{S}$ a relation over $sort(R)$. A database can also be viewed as a finite set of ground atoms using only the relation names in $\mathbf{S}$.

### 1.1.2 First-Order as Query Language

A database does not only provide a structure to store information but also provides ways to specify queries. Querying a database is a common task that can be achieved in a large variety of languages. Practically, people use languages like *SQL* (Structured Query Language), but in the theoretical field, first-order logic and its variants are more common.

*First-order logic* is a logic allowing the use of quantified variables. The adjective "first-order" distinguishes this logic from more expressive, higher-order logics which, for example, allow quantifications over subsets of the domain. In first-order logic, the quantification ranges over elements of the domain.

A *formula* in first-order logic is built from elements of a first-order language. If $\mathbf{S}$ is a database schema, then $\mathbf{S}$ determines a *first-order language* $\mathcal{L}(\mathbf{S})$ where the *$n$-ary predicate* symbols refer to the relation names in $\mathbf{S}$, that is, the $n$-ary relation symbol $R$ refers to the relation name $R$ with arity $n$.

Variables and their occurrences may be either *bound* or *free* in formulas. An occurrence of variable $x$ in a formula is bound if it occurs within the scope of $\exists x$ or $\forall x$; otherwise the occurrence is free. A free variable has at least one free occurrence. If a formula has no free variable occurrence, it is called a *first-order sentence.* If a formula has no variable, then it is called a *ground* formula.

**Example 1.2** Here are examples of two well-formed first-order formulas. Let $R$ be a relation name whose arity is 2. $x$ and $y$ are variables while $a, b$ and $c$ denote three constants.

$$\begin{array}{lcl} \psi_1 & : & \exists x\big(R(a,x) \wedge R(x,b)\big) \\ \psi_2 & : & \exists x\big(x = c \wedge R(x,y)\big) \end{array}$$

The variable $x$ is bound in $\psi_1$ and $\psi_2$. The variable $y$ is free in $\psi_2$. The formula $\psi_1$ is a first-order sentence as it has no free occurrence of variables. $\lhd$

In this document, we assume that the set of formulas and their interpretations over a given first-order language are the usual ones [Libkin 2004]. It is notwithstanding important to point out some particularities when considering first-order logic in the database field [Reiter 1982]:

- Function symbols are not included.
- One makes no clear distinction between constant symbols in the vocabulary and elements of the universe. Every constant symbol is interpreted by itself.

A *query* $q$ over schema $\mathbf{S}$ is a first-order formula $\psi$ in $\mathcal{L}(\mathbf{S})$. If $q$ has no free variable (i.e. $\psi$ is a first-order sentence), $q$ is called a *Boolean query.* If $q$ is a Boolean query and $\mathsf{db}$ an instance of $\mathbf{S}$, then we say that $\mathsf{db}$ satisfies $q$, denoted $\mathsf{db} \models q$, if $q$ evaluates to true on $\mathsf{db}$.

The answer to a nonBoolean query is defined using the notion of valuation. A *substitution* is a mapping $\sigma : \mathbf{var} \cup \mathbf{dom} \to \mathbf{var} \cup \mathbf{dom}$ such that for every constant

$a \in \mathbf{dom}$, $\sigma(a) = a$. A *valuation $\theta$ over $X \subseteq$* $\mathbf{var}$ is a substitution such that for every variable $x \in X$, $\theta(x) \in \mathbf{dom}$ ; if $x$ is a constant or a variable not in $X$, then $\theta(x) = x$. If $\vec{x} = \langle x_1, \ldots, x_n \rangle$ and $\theta$ is a substitution, we write $\theta(\vec{x})$ for $\langle \theta(x_1), \ldots, \theta(x_n) \rangle$.

Let $q$ be a query with free variables $\vec{x} = \langle x_1, \ldots, x_n \rangle$. The answer to $q$ on a database $\mathsf{db}$, denoted by $q(\mathsf{db})$, is the set:

$$q(\mathsf{db}) = \{\theta(\vec{x}) \mid \theta \text{ is a valuation over } x_1, \ldots, x_n \text{ such that } \mathsf{db} \models q\big(\theta(\vec{x})\big)\}.$$

Assume some fixed total order on the set of variables $\mathbf{var}$. This total order will only serve to "serialize" sets of variables into sequences in a unique way. If $\vec{x}$ is a sequence of variables and constants, then $\mathsf{vars}(\vec{x})$ is the set of variables that occur in $\vec{x}$. Let $q$ be a query, $\vec{x} = \langle x_1, \ldots, x_n \rangle$ a sequence of variables and $\vec{c} = \langle c_1, \ldots, c_n \rangle$ a sequence of constants. We denote by $q_{[\vec{x} \mapsto \vec{c}]}$ the query $q$ in which, for $1 \leq i \leq n$, every occurrence of $x_i$ is substituted by $c_i$.

We now introduce the subclass of conjunctive queries. A *conjunctive query $Q$* is a pair $Q = (q, V)$ where $q = \{R_1(\vec{x}_1), \ldots, R_n(\vec{x}_n)\}$ is a finite set of atoms and where $V$ is a subset of the variables occurring in $q$. Every variable of $V$ is free; the other variables are bound. This query represents the first-order formula $\exists u_1 \ldots \exists u_k \big(R_1(\vec{x}_1) \wedge \cdots \wedge R_n(\vec{x}_n)\big)$, where $u_1, \ldots, u_k$ are all the variables of $\mathsf{vars}(\vec{x}_1 \ldots \vec{x}_n) \setminus V$.

**Example 1.3** Query $\exists x(R(x, y) \wedge S(y, z))$ is represented as $(\{R(x, y), S(y, z)\}, \{y, z\})$. $\lhd$

Let $Q = (q, V)$ be a conjunctive query and $\mathsf{db}$ a database instance. Let $\vec{x}$ be the variables of $V$ ordered according to the total order on the set of variables. The answer to $Q$ on $\mathsf{db}$, denoted $Q(\mathsf{db})$, is defined as follows:

$$Q(\mathsf{db}) = \{\theta(\vec{x}) \mid \theta \text{ is a valuation over } \mathsf{vars}(q) \text{ such that } \theta(q) \subseteq \mathsf{db}\}.$$

We say that a conjunctive query $Q = (q, V)$ has a *self-join* if some relation name occurs more than once in $q$; if $Q$ has no self-join, then it is called *self-join-free*. The class of self-join-free conjunctive queries is denoted by **SJFC**.

Finally, if $V$ is empty, then $Q$ is a Boolean conjunctive query and then either $Q(\mathsf{db}) = \{\langle\rangle\}$ (representing **true**) or $Q(\mathsf{db}) = \{\}$ (representing **false**). If it is clear from the context that $V$ is empty, we will sometimes write $q$ to refer to $Q = (q, V)$.

## 1.2 Inconsistency and Uncertainty in Databases

### 1.2.1 Integrity Constraints

Reducing redundancy and improving reliability of the data are part of the database approach. Attributes and relation names are often chosen in a way to reflect some restrictions on the data. For example, one could expect that values used for attribute age are non negative integers or that every company stored in relation Emp has a location in relation Comp. Those *a priori* knowledges of restrictions or constraints on the values are essential in databases [Maier 1983]. Integrity constraints aim to formally integrate these knowledges into databases.

An *integrity constraint* is a first-order sentence that the database has to satisfy. An integrity constraint could encode several useful conditions and restrictions for the data. In this thesis, we focus on functional dependencies, and in particular on primary key constraints, a common class of integrity constraints. Other common integrity constraints are described in Chapters 7 and 14 in [Maier 1983].

**Definition 1.1** [Abiteboul et al. 1995] If $U$ is a set of attributes, then a *functional dependency* (FD) over $U$ is an expression of the form $X \to Y$, where $X, Y \subseteq U$. A relation $\mathsf{R}$ over $U$ satisfies $X \to Y$, denoted $\mathsf{R} \models X \to Y$, if for all tuples $p, q$ in $\mathsf{R}$, $p[X] = q[X]$ implies $p[Y] = q[Y]$. $\triangleleft$

Let $\mathbf{S}$ be a database schema. We associate with each relation name $R$ in $\mathbf{S}$ a unique *primary key* $X \subseteq sort(R)$. Assume a fixed database and let $\mathsf{R}$ be the relation associated with $R$. We say that relation $\mathsf{R}$ satisfies this primary key if $\mathsf{R}$ satisfies the functional dependency $X \to sort(R)$.

A *signature* for $R$ is a pair $[n, k]$ with $1 \leq k \leq n$ where $n$ is the arity of $R$ and $k$ defines a primary key on the first $k$ attributes of $R$: if $U = \{U_1, \ldots, U_n\}$ is the set of attributes of $R$, then a relation $\mathsf{R}$ for $R$ satisfies the primary key if $\mathsf{R}$ satisfies $\{U_1, \ldots, U_k\} \to U$. If $R$ has signature $[n, k]$, we write $R(\underline{\vec{x}}, \vec{y})$ for atom $R(a_1, \ldots, a_n)$, with $\vec{x} = a_1, \ldots, a_k$ and $\vec{y} = a_{k+1}, \ldots, a_n$, i.e. the primary key positions are underlined. If $n = k$, then $R$ is said to be *all-key*. Note that, in general, any attribute could be part of the primary key constraint; the restriction to the first $k$ attributes is only for convenience.

In the remainder, whenever the database is fixed, when we speak about relation $R$, we mean the relation associated with relation name $R$.

**Example 1.4** The relation $\mathsf{Emp}$ of $\mathsf{db}_1$ has signature $[3, 2]$. Relation $\mathsf{Comp}$ has signature $[2, 1]$. The underlined attributes in Figure 1.1 indicate the primary key for each relation. The primary key $\{\mathsf{Name}, \mathsf{WorksFor}\}$ on $\mathsf{Emp}$ states every employee has a unique age while the primary key $\{\mathsf{Name}\}$ on $\mathsf{Comp}$ states every company has a unique location. $\triangleleft$

## 1.2.2 Inconsistency by Constraints Violations

A database is said to be *consistent* if it satisfies a given set of integrity constraints. It is common to require databases to be consistent in data management. It is generally desirable that an evolving database is kept consistent during its whole life.

*Inconsistency* is inherent in many database applications and is generally an undesirable property of a database. For several reasons, when some integrity constraints are not satisfied, a database may be *inconsistent* and may contain inconsistent data. Inconsistency is not always avoidable. For example, inconsistency arises as an inconvenient but inescapable consequence of data integration and data exchange [Bertossi 2011].

**Example 1.5** Let $\mathsf{db}_{\mathsf{inc}}$ be the database of Figure 1.2. This database is inconsistent because the primary key constraint $\{\mathsf{Name}\}$ is not satisfied for tuples involving Microsoft in $\mathsf{Comp}$. The relation $\mathsf{Comp}$ of this database could have been obtained by combining the two data sources of Figure 1.3 containing $\mathsf{Comp}_{\mathsf{A}}$ and $\mathsf{Comp}_{\mathsf{B}}$. Note that the primary key is satisfied in both $\mathsf{Comp}_{\mathsf{A}}$ and $\mathsf{Comp}_{\mathsf{B}}$.

$\triangleleft$

| Emp | Name | WorksFor | Age |
|---|---|---|---|
| | Paul | Microsoft | 60 |
| | Bill | Microsoft | 57 |
| | Larry | Google | 40 |
| | Tim | Apple | 52 |

| Comp | Name | Location |
|---|---|---|
| | Microsoft | Redmond |
| | Microsoft | Cupertino |
| | Google | Mountain View |
| | Apple | Cupertino |

Figure 1.2: An inconsistent database $\mathsf{db}_{\mathsf{inc}}$, the primary key is violated in Comp.

| $\mathsf{Comp}_{\mathsf{A}}$ | Name | Location |
|---|---|---|
| | Microsoft | Cupertino |
| | Apple | Cupertino |

| $\mathsf{Comp}_{\mathsf{B}}$ | Name | Location |
|---|---|---|
| | Microsoft | Redmond |
| | Google | Mountain View |
| | Apple | Cupertino |

Figure 1.3: Two potential data sources for relation Comp.

Inconsistency is sometimes voluntary and is not necessarily a bad thing. In planning databases, for example, it seems natural to encode uncertainty using primary key violations. This way, one can represent different alternatives. Consider the planning database of Figure 1.4. This database stores information about the preferences for the location of world events. The exact location of Olympic Games 2020 is still uncertain, on the other hand, we are sure it will occur in Europe, even though we do not know the exact country: it can be in Germany or in Belgium.

### 1.2.3 Repairs and Consistent Answers

While it is sometimes possible to remove inconsistency by cleaning the database using some *data cleaning* techniques or by adding or removing tuples in order to satisfy the integrity constraints, this process is often complex and non-deterministic. Removing tuples in which the primary key is violated can lead to the loss of some useful information if the tuples consist of both correct and erroneous components.

A database that is inconsistent may still contain reliable information. For example, not all the tuples are involved in some primary key violation and the tuples that are not involved in primary key constraints may contain consistent data that can be used when querying the database. Tuples involved in primary key violations may also contain useful pieces of data.

| Events | Name | Year | Country |
|---|---|---|---|
| | Olympic Games | 2020 | Germany |
| | Olympic Games | 2020 | Belgium |

| Countries | Name | Continent |
|---|---|---|
| | Germany | Europe |
| | France | Europe |
| | Belgium | Europe |

Figure 1.4: An events planning database $\mathsf{db}_{\mathsf{plan}}$.

**Example 1.6** Consider database $\mathsf{db}_{\mathsf{plan}}$ of Figure 1.4. The relation $\mathsf{Countries}$ satisfies its primary key. Relation $\mathsf{Events}$ violates the primary key $\{\mathsf{Name}, \mathsf{Year}\}$ but it is still reliable that Olympic Games will be held in 2020. Moreover, it is certain that they will take place in Europe. $\triangleleft$

The existence of both consistent and inconsistent data leads us to characterize what is consistency and what is the answer that is certain for a given query. Let us first introduce the notion of *repairs* of an inconsistent database. The symmetric difference $\Delta$ is used to capture the distance between two databases. Let $\mathsf{db}$ and $\mathsf{db}'$ be two databases, then $\Delta(\mathsf{db}, \mathsf{db}') = (\mathsf{db} \setminus \mathsf{db}') \cup (\mathsf{db}' \setminus \mathsf{db})$.

**Definition 1.2** [Arenas et al. 1999] Let $\mathcal{IC}$ be a set of integrity constraints. A *repair* of a (possibly) inconsistent database $\mathsf{db}$ is a database $\mathsf{rep}$ over the same schema such that $\mathsf{rep}$ satisfies $\mathcal{IC}$ and, for every database $\mathsf{rep}'$ such that $\mathsf{rep}'$ satisfies $\mathcal{IC}$, $\Delta(\mathsf{db}, \mathsf{rep}) \subseteq \Delta(\mathsf{db}, \mathsf{rep}')$. $\triangleleft$

Intuitively, a repair of an inconsistent database is a (consistent) database that minimally differs from the inconsistent database. This notion of repair relies on the symmetric difference between two databases. Alternative notions of repair have also been investigated in the literature. For example, [Arenas et al. 2003; Lopatenko and Bertossi 2006; Afrati and Kolaitis 2009] consider the cardinality of the symmetric difference while [Chomicki and Marcinkowski 2005] considers repairs that only allow tuple deletions. Several attribute-based repairs have also been considered by [Bertossi et al. 2008; Wijsen 2005] where the repairs are obtained by modifying the attribute values in the tuples. [Lopatenko and Bertossi 2006] gives some comparisons of different repair semantics. In this thesis however, we work with the notion of repair of [Arenas et al. 1999] of Definition 1.2 which is the most commonly considered in the literature [Staworko and Chomicki 2010].

Repairs are useful to characterize the answer that is consistent in a database: the consistent answer to a query is the answer to this query that is invariant regardless of the chosen repair.

**Definition 1.3** [Arenas et al. 1999] The *consistent answer* of a query $q$ over a (possibly) inconsistent database $\mathsf{db}$, denoted $q_{\mathsf{sure}}(\mathsf{db})$, is the intersection of the answers to $q$ on every repair of $\mathsf{db}$:

$$q_{\mathsf{sure}}(\mathsf{db}) = \bigcap \{q(\mathsf{rep}) \mid \mathsf{rep} \text{ is a repair of } \mathsf{db}\}.$$

If $q$ is a Boolean query, then we say that $q$ is *consistently true* in $\mathsf{db}$ if and only if $q$ evaluates to true on every repair of $\mathsf{db}$, denoted by $\mathsf{db} \models_{\mathsf{sure}} q$. $\triangleleft$

**Example 1.7** The databases $\mathsf{rep}_1$ and $\mathsf{rep}_2$ in Figure 1.5 are the repairs of the uncertain planning database $\mathsf{db}_{\mathsf{plan}}$ of Figure 1.4. Consider the following Boolean conjunctive queries $q_3$ and $q_4$ asking if the Olympic Games will be held some year in Europe ($q_3$) or in Belgium ($q_4$).

$$\begin{aligned} q_3 &= \exists x \exists y \big(\mathsf{Events}(\underline{\text{‘Olympic Games’}, x}, y) \land \mathsf{Countries}(\underline{y}, \text{‘Europe’})\big). \\ q_4 &= \exists x \exists y \big(\mathsf{Events}(\underline{\text{‘Olympic Games’}, x}, \text{‘Belgium’})\big). \end{aligned}$$

$\mathsf{rep}_1 =$

| Events | Name | Year | Country |
|---|---|---|---|
| | Olympic Games | 2020 | Germany |

| Countries | Name | Continent |
|---|---|---|
| | Germany | Europe |
| | France | Europe |
| | Belgium | Europe |

$\mathsf{rep}_2 =$

| Events | Name | Year | Country |
|---|---|---|---|
| | Olympic Games | 2020 | Belgium |

| Countries | Name | Continent |
|---|---|---|
| | Germany | Europe |
| | France | Europe |
| | Belgium | Europe |

Figure 1.5: The two repairs of $\mathsf{db}_{\mathsf{plan}}$.

The consistent answer to $q_3$ is true on $\mathsf{db}_{\mathsf{plan}}$ because $\mathsf{rep}_1$ and $\mathsf{rep}_2$ both satisfy $q_3$ (using respectively 'Germany' and 'Belgium' for $y$). In contrast, the consistent answer to $q_4$ is not true on $\mathsf{db}_{\mathsf{plan}}$ because $q_4$ evaluates to false on $\mathsf{rep}_1$. Note that both $q_3$ and $q_4$ evaluate to true on $\mathsf{db}_{\mathsf{plan}}$.

$\triangleleft$

Computing the consistent answer to a query on an inconsistent database is usually referred to as the *consistent query answering* problem. Given a fixed query $q$ and an input database $\mathsf{db}$, the problem of consistent query answering is the problem of computing the consistent answer to $q$ on $\mathsf{db}$. Notice that $q$ is not part of the input, so the complexity of the problem is data complexity.

The complexity of consistent query answering has been studied for several classes of integrity constraints, for example, for primary key constraints [Arenas et al. 1999; Fuxman and Miller 2005; Wijsen 2012], for denial constraints [Chomicki et al. 2004], for binary universal constraints [Celle and Bertossi 2000], and for inclusion dependencies [Calì 2005]. Elaborated surveys can be found in [Chomicki and Marcinkowski 2004; Bertossi 2006] and in [Bertossi 2011].

In the remaining of this thesis, we consider that the set $\mathcal{IC}$ of integrity constraints is only composed of primary key constraints. The terminology is slightly adapted to reflect the focus on primary key constraints and uncertainty. In this context, we say that a database is *uncertain* if the database contains two distinct tuples of the same relation that share the same primary key value. An uncertain database $\mathsf{db}$ gives rise to a set of repairs. Every repair of an uncertain database is then obtained by selecting a maximal number of tuples without ever selecting two distinct tuples of the same relation that agree on their primary key. We use the term *certain answer* to a query $q$ on $\mathsf{db}$ to refer to the consistent answer to $q$ on $\mathsf{db}$. If $q$ is Boolean, then we say that $q$ is *certain* in $\mathsf{db}$ if it evaluates to true on every repair of $\mathsf{db}$.

If the set of integrity constraints is restricted to primary key constraints, the problem of computing the certain answer is known as the *certain query answering* problem. For every fixed conjunctive query, the certain query answering problem is in **coNP**. Depending on the query, however, the complexity may vary[1]: for instance, for the Boolean query

[1]The classes **coNP** and $\mathbf{AC^0}$ are covered, among others, in [Papadimitriou 1994].

$\exists x \exists y \exists z \big(R(\underline{x}, z) \wedge S(\underline{y}, z)\big)$, the problem is **coNP**-complete [Chomicki and Marcinkowski 2005] while it is in **P** (and even in $\mathbf{AC^0}$) for the Boolean query $\exists x \exists y \exists z \big(R(\underline{x}, y) \wedge S(\underline{y}, z)\big)$.

### 1.2.4 Certain First-Order Rewriting

Let $q$ be a Boolean query. To see if $q$ is certain in some database $\mathsf{db}$, it is not always needed to compute the repairs of $\mathsf{db}$ and to evaluate $q$ on each of them. One can compute a first-order query $\varphi$ such that $\varphi$ is true in $\mathsf{db}$ if and only if $q$ evaluates to true on every repair of $\mathsf{db}$. If $\varphi$ exists, it is called a *certain first-order rewriting for q*.

The interest in certain first-order rewriting is evident: if there exists a first-order sentence to compute the certain answer of a query $q$ on an input database $\mathsf{db}$, then the problem of computing the certain answer of $q$ can be solved in the low complexity class $\mathbf{AC^0}$ and encoded in classical database technologies. Certain first-order rewriting avoids the computation of the (possibly exponentially many [Chomicki and Marcinkowski 2005]) repairs and can be used directly on the uncertain database to compute the certain answer to a query.

**Example 1.8** There is no need to evaluate $q_3$ on the repairs in Figure 1.5 to see that $q_3$ is certain. It suffices to check if the following first-order sentence $\psi_3$ evaluates to true on the original database:

$$\begin{aligned}\psi_3 \;=\; & \exists x \exists y \Big( \mathsf{Events}(\underline{\text{`Olympic Games'}, x}, y) \wedge \forall y' \Big( \mathsf{Events}(\underline{\text{`Olympic Games'}, x}, y') \\ & \rightarrow \big(\mathsf{Countries}(\underline{y'}, \text{`Europe'}) \wedge \forall z (\mathsf{Countries}(\underline{y'}, z) \rightarrow z = \text{`Europe'})\big)\Big)\Big).\end{aligned}$$

◁

**Definition 1.4** A *certain first-order rewriting* for a query $q(\vec{x})$ is a first-order query $\varphi(\vec{x})$ such that for every database $\mathsf{db}$, for every $\vec{a} \in \mathbf{dom}^{|\vec{x}|}$, we have

$$\mathsf{db} \models \varphi(\vec{a}) \iff \langle \vec{a} \rangle \in q_{\mathsf{sure}}(\mathsf{db}).$$

◁

In particular, if $q$ is a Boolean query, a certain first-order rewriting for $q$ is a first-order query $\varphi$ such that for every database $\mathsf{db}$, $\varphi$ evaluates to true on $\mathsf{db}$ if and only if $q$ evaluates to true on every repair of $\mathsf{db}$.

Figure 1.6 illustrates the process of certain first-order rewriting. Uncertain database $\mathsf{db}$ (top left) gives rise to a set of repairs (bottom left). Computing the certain answer (top right) of a given query $q$ can be done by computing the answer to $q$ on every possible repair of $\mathsf{db}$ (bottom right) and by taking the intersection of those answers. A first-order query $\varphi$ is a certain query rewriting for $q$ if, given $\mathsf{db}$, the certain answer to $q$ can be computed directly on $\mathsf{db}$. The problem is the following: given a first-order query $q$, can we decide if such a formula $\varphi$ exists? And if it exists, can we compute $\varphi$?

The remaining of this thesis is split in two parts:

$$\begin{tikzcd}
\mathsf{db} \arrow[rr, bend left] \arrow[d] & \text{certain FO rew. } \varphi & q_{\mathsf{sure}}(\mathsf{db}) = \varphi(\mathsf{db}) \\
\{\mathsf{rep} \text{ is a repair of } \mathsf{db}\} \arrow[rr, bend right] & q \arrow[u, dashed] & \{q(\mathsf{rep}) \mid \mathsf{rep} \text{ is a repair of } \mathsf{db}\} \arrow[u, "\bigcap"]
\end{tikzcd}$$

Figure 1.6: Query $\varphi$ is a first-order rewriting for query $q$.

## Part 1. Certain Conjunctive Query Answering in SQL

In the first part of this thesis, we will be interested in computing the certain answer to a query in SQL and, as a first step, in first-order logic.

Chapter 2 presents the problem $\mathsf{CERTAINTY}(q)$ which, given a Boolean query $q$, is the set of (possibly) uncertain databases such that $q$ is certain in the database. Defining $\mathsf{CERTAINTY}(q)$ in first-order logic is equivalent to saying that $q$ has a certain first-order rewriting. Recently, a syntactic characterization was obtained for the class of acyclic self-join-free conjunctive queries for which the certain answer is definable by a first-order formula [Wijsen 2012]. This characterization relies on a tool called the attack graph.

Based on these results, we propose in Chapter 3 a procedure that, given an acyclic Boolean self-join-free conjunctive query $q$ whose attack graph is acyclic, computes a certain first-order rewriting $\varphi$ of $q$. We show that, if we translate in SQL those certain first-order rewritings computed using this procedure, the resulting queries often contain deeply nested subqueries which can lead to poor performance on practical databases. We study the nesting and alternation of quantifiers in certain first-order rewritings, and propose two syntactic simplification techniques that aim to reduce these metrics. Such simplification techniques impact the number of (nested) subqueries in the certain SQL rewritings and could improve the performance when evaluating the queries.

In Chapter 4, we show how to use those results to compute "improved" certain SQL rewritings. We then investigate whether these syntactic simplifications systematically result in lower execution times on SQL databases. We also experimentally study what happens when we increase the number of tuples involved in primary key violations, or the number of tuples in the database, or the length of the query.

Chapter 5 concludes the first part.

## Part 2. Certain Answers in Uncertain Database Histories

The second part of the thesis deals with uncertainty in the framework of first-order logic on words. In this setting, uncertainty is captured by the concept of multiword, a finite sequence of nonempty sets of possible symbols. Every word obtained by selecting one symbol from each set of possible symbols is a possible word. We see that this context leads us to consider a variant of a pattern matching problem: given a word $w$, is $w$ a factor of every possible word represented by a multiword.

In Chapter 6, we situate our variant of the pattern matching problem with multiwords among several other variants. We define the concept of multiwords and the language $\mathsf{CERTAIN}(w)$. Given a word $w$, $\mathsf{CERTAIN}(w)$ is the set of all multiwords such that $w$ is a factor of every possible word of the multiword. We see how a problem of certain query answering in database histories can be viewed as an application of the pattern matching problem with multiwords.

In Chapter 7, we are interested in the first-order definability of $\mathsf{CERTAIN}(w)$ and we postulate the conjecture that $\mathsf{CERTAIN}(w)$ is first-order definable for every word $w$. This conjecture has been experimentally checked for a very large set of words. We show that $\mathsf{CERTAIN}(w)$ is regular and that its first-order definability is equivalent to its aperiodicity.

In Chapter 8, we provide strong supporting evidence for the conjecture that the language $\mathsf{CERTAIN}(w)$ is first-order definable. We introduce several families of words $w$ such that $\mathsf{CERTAIN}(w)$ is aperiodic, and we study those large families of words.

In Chapter 9, we study deterministic finite automata for $\mathsf{CERTAIN}(w)$. We study the minimal deterministic finite automata recognizing $\mathsf{CERTAIN}(w)$ and we focus on the size of those minimal automata.

Chapter 10 concludes this second part.

# Part I

# Certain Conjunctive Query Answering in SQL

CHAPTER 2

# The Problem of Certain Answers

An uncertain database is defined as a database in which primary keys need not be satisfied. The certain answer to a query $q$ on an uncertain database $\mathsf{db}$, denoted $q_{\mathsf{sure}}(\mathsf{db})$, is defined as the intersection of the answers to $q$ on every repair $\mathsf{rep}$ of $\mathsf{db}$.

We are interested in computing the certain answer to a first-order query $q$ by means of a technique known as certain first-order rewriting. This technique avoids the computation of the repairs: to know whether $q$ is true in every repair, it suffices to execute a certain first-order rewriting for $q$ once on the original database. Moreover, if there exists a first-order sentence to compute the certain answer of a query $q$ on an input database $\mathsf{db}$, then the problem of computing the certain answer of $q$ can be solved in $\mathbf{AC^0}$ and encoded in classical database technologies.

We define in Section 2.1 the set $\mathsf{CERTAINTY}(q)$, where $q$ is a Boolean query, as the set of databases in which $q$ is certain, meaning $q$ evaluates to true on every repair of the database. The first-order definability of $\mathsf{CERTAINTY}(q)$ will be the focus of the first part of the thesis.

If $q$ is an acyclic Boolean self-join-free conjunctive query (**SJFC**), then the first-order definability of $\mathsf{CERTAINTY}(q)$ is decidable [Wijsen 2010]. The decidability relies on a tool called the *attack graph*. The attack graph of $q$ is acyclic if and only if $\mathsf{CERTAINTY}(q)$ is first-order definable. This result is explained in Section 2.2. In Section 2.3, we generalize known results for the first-order definability of $\mathsf{CERTAINTY}(q)$ for a restricted class of cyclic self-join-free conjunctive queries.

## 2.1 The Problem $\mathsf{CERTAINTY}(q)$

For a given Boolean query $q$, the decision problem $\mathsf{CERTAINTY}(q)$ is the problem that takes as input an uncertain database $\mathsf{db}$ and asks whether the Boolean query $q$ evaluates to true on every repair.

**Definition 2.1** Let $q$ be a Boolean query. We define the following set:

$$\mathsf{CERTAINTY}(q) = \{\mathsf{db} \mid q \text{ is true in every repair of database } \mathsf{db}\}.$$

◁

$\mathsf{CERTAINTY}(q)$ is first-order expressible if there exists a first-order sentence $\varphi$ such that, for every database $\mathsf{db}$, $\mathsf{db} \in \mathsf{CERTAINTY}(q)$ if and only if $\varphi$ evaluates to true on $\mathsf{db}$. The formula $\varphi$, if it exists, is called a certain first-order rewriting for $q$. The interest in certain first-order rewriting is evident. Since $\varphi$ is first-order, it can be efficiently evaluated using a database engine: one can encode $\varphi$ in SQL and then execute the resulting SQL query in polynomial time data complexity using standard database technology. Note that the restriction to Boolean queries simplifies the technical treatment, but is not fundamental (see page 22).

Certain (or consistent) query answering was founded in the seminal work by Arenas, Bertossi, and Chomicki [Arenas et al. 1999]. The current state of the art can be found in [Bertossi 2011].

The non-existence of a certain first-order rewriting can be settled by complexity-theoretic arguments: if $\mathsf{CERTAINTY}(q)$ is **coNP**-hard, then it is not first-order expressible (see, for example, [Chomicki and Marcinkowski 2005; Fuxman and Miller 2007]). It is important to note that the decision problem $\mathsf{CERTAINTY}(q)$ is about the *data complexity* of certain query answering: the query (and the schema, including the integrity constraints) is fixed and the complexity of computing the certain answer depends only on the size of the database.

It is already known that there exists a simple Boolean conjunctive query $q$ with only two atoms for which the problem $\mathsf{CERTAINTY}(q)$ is **coNP**-complete [Fuxman and Miller 2007]. The complexity of $\mathsf{CERTAINTY}(q)$ for conjunctive queries $q$ has been widely studied, also outside the database community [Bienvenu 2012].

[Fuxman and Miller 2007] were the first ones to focus on the first-order definability of $\mathsf{CERTAINTY}(q)$, with applications in the ConQuer system [Fuxman et al. 2005]. They introduced a class of conjunctive queries without self-join, called $\mathcal{C}_{\mathit{forest}}$. Every query in this class has a certain first-order rewriting. However, it was shown that the query $q_0 = \exists x \exists y (R(\underline{x}, y) \wedge S(\underline{x}, y))$ is not in $\mathcal{C}_{\mathit{forest}}$ but has a certain first-order rewriting.

Their results have been generalized by [Wijsen 2009a] who presented a larger class of queries (including $q_0$) that admit a certain first-order rewriting. In [Wijsen 2009b], a semantic class called $\mathcal{C}_{\mathit{rooted}}$ was defined. This class contains conjunctive queries $q$ such that $\mathsf{CERTAINTY}(q)$ is first-order expressible. Queries in this class can be cyclic and can contain self-joins. Unfortunately, no membership test for $\mathcal{C}_{\mathit{rooted}}$ was provided.

[Wijsen 2010, 2012] studied the class of self-join-free conjunctive (**SJFC**) queries that are acyclic and presented a significant breakthrough by giving a characterization of the queries in this class that have a certain first-order rewriting. We will recall those results in the next section. It follows from [Wijsen 2012] that if an acyclic **SJFC** query has a certain first-order rewriting, then it belongs to $\mathcal{C}_{rooted}$.

The class of acyclic **SJFC** queries which are rewritable and the class $\mathcal{C}_{forest}$ are incomparable under set inclusion: there exist acyclic first-order rewritable conjunctive queries not in $\mathcal{C}_{forest}$ and the class $\mathcal{C}_{forest}$ contains some cyclic first-order rewritable queries. However, Theorem 2.3 on page 23 identifies a class $\mathcal{C}$ of **SJFC** queries such that (1) every query in $\mathcal{C}$ has a certain first-order rewriting, (2) $\mathcal{C}$ contains $\mathcal{C}_{forest}$, and (3) $\mathcal{C}$ contains all acyclic **SJFC** queries with an acyclic attack graph.

The class of acyclic **SJFC** queries is a large class of practical interest. The restriction to queries without self-join is quite usual in uncertain [Fuxman and Miller 2007] and probabilistic databases [Dalvi and Suciu 2007]. It is important to note that relatively little is known for queries that are cyclic and/or contain self-joins. [Wijsen 2009b] showed that, for $q = \exists x \exists y \big(R(\underline{x}, y) \land R(\underline{y}, a)\big)$ (with self-join), $\mathsf{CERTAINTY}(q)$ is in **P** but not first-order expressible.

It is an open conjecture that, for every conjunctive query $q$ without self-join, either $\mathsf{CERTAINTY}(q)$ is in **P** or **coNP**-complete. This dichotomy result has been recently shown to be true for queries with exactly two atoms [Kolaitis and Pema 2012].

The counting variant of $\mathsf{CERTAINTY}(q)$, denoted $\natural\mathsf{CERTAINTY}(q)$ was recently studied in [Maslowski and Wijsen 2011, 2013]. This variant takes as input an uncertain database and asks to determine the exact number of repairs that satisfy some Boolean query $q$. The authors showed that for every Boolean **SJFC** query $q$, $\natural\mathsf{CERTAINTY}(q)$ is in **P** or $\natural$**P**-complete[1]. The problem $\natural\mathsf{CERTAINTY}(q)$ is closely related to query answering in probabilistic data models (see, for example, [Dalvi et al. 2009]). From the probabilistic database angle, the problem $\mathsf{CERTAINTY}(q)$ is solved if we can determine whether query $q$ evaluates to probability 1 on the probabilistic database obtained from $\mathsf{db}$ by assuming a uniform probability distribution over the set of repairs of $\mathsf{db}$. The uncertain databases can be viewed as a restricted case of *block-independent-disjoint* probabilistic databases [Dalvi et al. 2009, 2011]. A *block* in a database $\mathsf{db}$ is a maximal subset of key-equal atoms. If $\{R(\underline{\vec{a}}, \vec{b}_1), \ldots, R(\underline{\vec{a}}, \vec{b}_n)\}$ is a block of size $n$, then every atom of the block has a probability of $1/n$ to be selected in a repair of $\mathsf{db}$. Every repair is a possible world, and all these worlds have the same probability.

## 2.2 Certain FO Rewriting for Acyclic SJFC Queries

Given a first-order query $q$, we are interested in the first-order definability of the set $\mathsf{CERTAINTY}(q)$ or, equivalently, in a certain first-order rewriting for $q$. We focus on a class of queries for which it has been shown that the first-order definability of $\mathsf{CERTAINTY}(q)$ is decidable. In [Wijsen 2012], it was shown that there exists a syntactic characterization of the frontier between first-order expressibility and inexpressibility of $\mathsf{CERTAINTY}(q)$ if $q$ is a Boolean self-join-free conjunctive (**SJFC**) query that is acyclic (in the sense of [Beeri et al. 1983]).

This syntactic characterization relies on a tool called the attack graph of $q$. This tool

[1]The class $\natural$**P** contains the counting variant of problems in **NP**.

is useful to characterize the queries that have a certain first-order rewriting and can also be used to compute such a certain first-order rewriting.

Subsection 2.2.1 introduces the notations and terminology. Subsection 2.2.2 details the construction of the attack graph of a query. Subsection 2.2.3 gives the main result: a sufficient and necessary condition to decide the existence of certain first-order rewritings for acyclic **SJFC** queries. Those three subsections are based on [Wijsen 2012].

### 2.2.1 Preliminaries

We now introduce some terminology, notations and notational conventions. Letters $F, G, H, J$ will be used for atoms appearing in a query. For $F = R(\underline{\vec{x}}, \vec{y})$, we denote by $\mathsf{keyVars}(F)$ the set of variables that occur in $\vec{x}$, and by $\mathsf{vars}(F)$ the set of variables that occur in $F$, that is, $\mathsf{keyVars}(F) = \mathsf{vars}(\vec{x})$ and $\mathsf{vars}(F) = \mathsf{vars}(\vec{x}) \cup \mathsf{vars}(\vec{y})$.

**Definition 2.2** A *join tree* for a conjunctive query $q$ is an undirected tree whose vertices are the atoms of $q$ such that the following condition is satisfied:

> *Connectedness Condition:* whenever the same variable $x$ occurs in two atoms $F$ and $G$, then $x$ occurs in each atom on the unique path linking $F$ and $G$.

An edge between atoms $F$ and $G$ is labelled by the set $L = \mathsf{vars}(F) \cap \mathsf{vars}(G)$ and is denoted by $F \overset{L}{\frown} G$. ◁

The term *Connectedness Condition* appears in [Gottlob et al. 2002] and refers to the fact that the set of vertices in which $x$ occurs induces a connected subtree. A conjunctive query $q$ is *acyclic* if it has a join tree. Notice that a query can have several distinct join trees. The notions of join tree and acyclicity are standard [Beeri et al. 1983].

Examples 2.1, 2.2 and 2.3 are borrowed from [Wijsen 2012].

**Example 2.1** Let $q_1 = \{R_0(\underline{x}, y), R_1(\underline{y}, x), R_2(\underline{x, y}), R_3(\underline{x}, z), R_4(\underline{x}, z)\}$. A join tree $\tau_1$ for $q_1$ is illustrated in Figure 2.1 (left). Consider query $\{S_0(\underline{y, z}, u), S_1(\underline{x}, y), S_2(\underline{z}, x, u)\}$. This query is cyclic and thus has no join tree. ◁

Every atom $F$ in a query $q$ induces a set of functional dependencies among the variables in $F$. As an example, $R(\underline{x, y}, z)$ induces the functional dependency $\{x, y\} \rightarrow \{x, y, z\}$. The set $\mathcal{K}(q)$ defined next contains every functional dependency that is induced by the atoms of $q$.

**Definition 2.3** Let $q$ be a Boolean conjunctive query. $\mathcal{K}(q)$ is defined as the following set of functional dependencies:

$$\mathcal{K}(q) = \{\mathsf{keyVars}(F) \rightarrow \mathsf{vars}(F) \mid F \in q\}.$$

◁

**Example 2.2** Let $q_2 = \{R_0(\underline{x}, y), R_1(\underline{y}, x), R_2(\underline{a}, x)\}$ be the query whose join tree $\tau_2$ is shown in Figure 2.2 (left). Then, $\mathcal{K}(q_2)$ contains $\{x\} \rightarrow \{x, y\}$, $\{y\} \rightarrow \{x, y\}$, and $\{\} \rightarrow \{x\}$. The latter functional dependency arises in the atom $R_2(\underline{a}, x)$ whose primary key position contains no variable. ◁

Recall from relational database theory [Ullman 1988, page 387] that if $\Sigma$ is a set of functional dependencies over a set $U$ of attributes and $X \subseteq U$, then the attribute closure of $X$ (with respect to $\Sigma$) is the set $\{A \in U \mid \Sigma \models X \rightarrow A\}$. If $X$ is equal to the closure of $X$, then $X$ is said to be closed.

**Definition 2.4** Let $q$ be a Boolean conjunctive query. For every atom $F$ in $q$, the set $F^{+,q}$ is defined as follows:

$$F^{+,q} = \{x \in \mathsf{vars}(q) \mid \mathcal{K}(q \setminus \{F\}) \models \mathsf{keyVars}(F) \rightarrow x\}.$$

$\lhd$

In words, $F^{+,q}$ is the attribute closure of the set $\mathsf{keyVars}(F)$ with respect to the set of functional dependencies that are induced by the atoms of $q \setminus \{F\}$. Note that variables play the role of attributes in this framework.

### 2.2.2 The Attack Graph

Let $q$ be an acyclic Boolean conjunctive query. For each join tree $\tau$, a new graph, called the attack graph of $\tau$, is computed. The vertices of a join tree and its attack graph are the same, but unlike join trees, attack graphs are directed graphs.

Attack graphs turn out to be a key tool to decide the existence of certain first-order rewritings for acyclic **SJFC** queries. The attack graph of a join tree $\tau$ for a query $q$ has a cycle if and only if $\mathsf{CERTAINTY}(q)$ is first-order definable [Wijsen 2012]. This is Theorem 2.2.

**Definition 2.5** Let $q$ be an acyclic Boolean conjunctive query. Let $\tau$ be a join tree for $q$. The *attack graph* of $\tau$ is a directed graph whose vertices are the atoms of $q$. There is a directed edge from $F$ to $G$ if $F, G$ are distinct atoms such that for every label $L$ on the unique path that links $F$ and $G$ in $\tau$, we have $L \nsubseteq F^{+,q}$. We write $F \overset{\tau}{\rightsquigarrow} G$ if the attack graph of $\tau$ contains a directed edge from $F$ to $G$. The directed edge $F \overset{\tau}{\rightsquigarrow} G$ is also called an *attack from $F$ to $G$*. If $F \overset{\tau}{\rightsquigarrow} G$, we say that $F$ *attacks* $G$ (or that $G$ is attacked by $F$). $\lhd$

Intuitively, the attack graph of a join tree for a query $q$ represents an "incidence link" between the atoms of $q$. These incidence links are induced by the dependencies that arise between the variables occurring at a key position in the atoms and the variables in the other atoms of $q$.

**Example 2.3** Let $\tau_2$ be the join tree for $q_2$ of Example 2.2. To simplify the notation, let $F = R_0(\underline{x}, y)$, $G = R_1(\underline{y}, x)$, and $H = R_2(\underline{a}, x)$, as indicated in Figure 2.2 (left).

The right part of Figure 2.2 shows the attack graph of $\tau_2$. We now explain how to construct this attack graph. We first compute $\mathcal{K}(q_2 \setminus \{F\}), \mathcal{K}(q_2 \setminus \{G\})$ and $\mathcal{K}(q_2 \setminus \{H\})$:

$$\begin{array}{rcl}
\mathcal{K}(q_2 \setminus \{F\}) & = & \{\{y\} \rightarrow \{x, y\}, \{\} \rightarrow \{x\}\} \\
\mathcal{K}(q_2 \setminus \{G\}) & = & \{\{x\} \rightarrow \{x, y\}, \{\} \rightarrow \{x\}\} \\
\mathcal{K}(q_2 \setminus \{H\}) & = & \{\{x\} \rightarrow \{x, y\}, \{y\} \rightarrow \{x, y\}\}
\end{array}$$

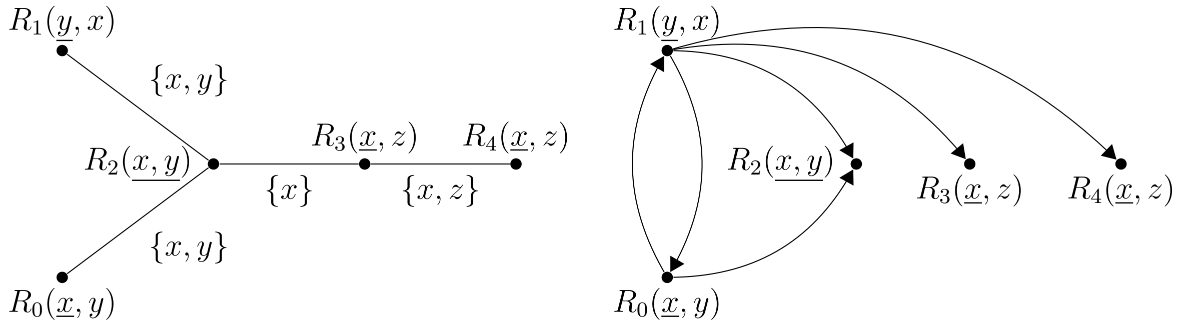

Figure 2.1: Join tree (left) and attack graph (right) for Boolean query $q_1$ of Example 2.1.

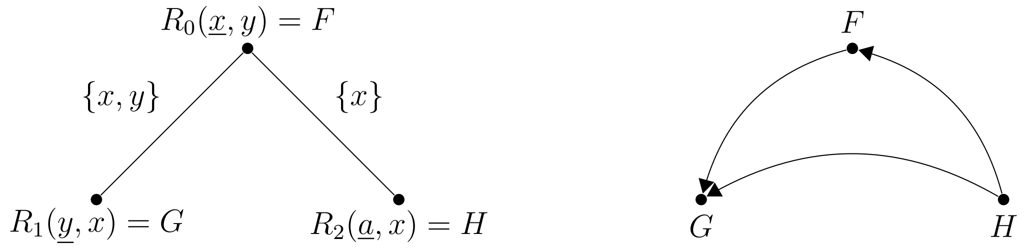
$R_0(\underline{x}, y) = F$
$\{x, y\}$
$\{x\}$
$R_1(\underline{y}, x) = G$
$R_2(\underline{a}, x) = H$
$F$
$G$
$H$

Figure 2.2: Join tree (left) and attack graph (right) for Boolean query $q_2$ of Example 2.2.

We have $\mathsf{keyVars}(F) = \{x\}$, which is already closed with respect to $\mathcal{K}(q_2 \setminus \{F\})$. Thus, $F^{+,q_2} = \{x\}$. The path from $F$ to $G$ in the join tree is $F \overset{\{x,y\}}{\frown} G$. Since the label $\{x, y\}$ is not contained in $F^{+,q_2}$, the attack graph contains a directed edge from $F$ to $G$, i.e. $F \overset{\tau_2}{\rightsquigarrow} G$. The path from $F$ to $H$ in the join tree is $F \overset{\{x\}}{\frown} H$. Since the label $\{x\}$ is contained in $F^{+,q_2}$, the attack graph contains no directed edge from $F$ to $H$.

We have $\mathsf{keyVars}(G) = \{y\}$ and the closure of $\{y\}$ with respect to $\mathcal{K}(q_2 \setminus \{G\})$ is $\{x, y\}$. Thus, $G^{+,q_2} = \{x, y\}$. The path from $G$ to $F$ in the join tree is $G \overset{\{x,y\}}{\frown} F$. Since the label $\{x, y\}$ is contained in $G^{+,q_2}$, the attack graph contains no directed edge from $G$ to $F$. For that same reason, the attack graph contains no directed edge from $G$ to $H$.

Finally, we have $\mathsf{keyVars}(H) = \{\}$, which is already closed with respect to $\mathcal{K}(q_2 \setminus \{H\})$. Thus, $H^{+,q_2} = \{\}$. The path from $H$ to $G$ in the join tree is $H \overset{\{x\}}{\frown} F \overset{\{x,y\}}{\frown} G$. Since no label on that path is contained in $H^{+,q_2}$, the attack graph contains a directed edge from $H$ to $G$, i.e. $H \overset{\tau_2}{\rightsquigarrow} G$. It is then obvious that the attack graph must also contain a directed edge from $H$ to $F$, i.e. $H \overset{\tau_2}{\rightsquigarrow} F$. $\lhd$

**Example 2.4** Let $q_1$ be the query of Example 2.1. Figure 2.1 shows the attack graph (right) of join tree $\tau_1$ (left). This attack graph contains a cycle. $\lhd$

An acyclic conjunctive query can have more than one join tree. We show in the next theorem that all these join trees have the same attack graph.

**Theorem 2.1** *Let $q$ be an acyclic Boolean conjunctive query. Let $\tau_1$ and $\tau_2$ be two join trees for $q$. The attack graphs of $\tau_1$ and $\tau_2$ are identical.*

Proof. Let $F, G$ be distinct atoms of $q$ such that $F \overset{\tau_1}{\rightsquigarrow} G$. We show $F \overset{\tau_2}{\rightsquigarrow} G$. The proof runs by induction on the size of the path between $F$ and $G$ in $\tau_1$. The base case is when this path is exactly $F \frown G$. Thus, the join tree $\tau_1$ contains an edge between $F$ and $G$. Since $F \overset{\tau_1}{\rightsquigarrow} G$, we can assume a variable $x \in \mathsf{vars}(F) \cap \mathsf{vars}(G)$ such that $x \notin F^{+,q}$. By the *Connectedness Condition*, the variable $x$ is an element of every label on the unique path between $F$ and $G$ in $\tau_2$. It follows $F \overset{\tau_2}{\rightsquigarrow} G$. For the induction step, assume the size of the path between $F$ and $G$ in $\tau_1$ is $k > 2$. Let $H$ be an element of this path such that $\tau_1$ contains an edge between $H$ and $G$. That is, $H$ is the atom on the path between $F$ and $G$ in $\tau_1$ such that $H$ is incident with $G$.

- Since $F \overset{\tau_1}{\rightsquigarrow} G$, we can assume a variable $x \in \mathsf{vars}(H) \cap \mathsf{vars}(G)$ such that $x \notin F^{+,q}$. By the *Connectedness Condition*, the variable $x$ is an element of every label on the unique path between $H$ and $G$ in $\tau_2$.
- By Lemma 4.9 in [Wijsen 2012], we have $F \overset{\tau_1}{\rightsquigarrow} G$ implies $F \overset{\tau_1}{\rightsquigarrow} H$. By the induction hypothesis, $F \overset{\tau_2}{\rightsquigarrow} H$. Thus, every label on the unique path between $F$ and $H$ in $\tau_2$ contains a variable that does not belong to $F^{+,q}$.

So, every label on the unique path between $F$ and $G$ in $\tau_2$ contains a variable that does not belong to $F^{+,q}$. It follows $F \overset{\tau_2}{\rightsquigarrow} G$. By symmetry, for all distinct atoms $F, G$ of $q$, $F \overset{\tau_2}{\rightsquigarrow} G$ implies $F \overset{\tau_1}{\rightsquigarrow} G$. Thus, the attack graphs of $\tau_1$ and $\tau_2$ contain the same directed edges. □

An important consequence of this theorem is that we can now unambiguously talk about *the* attack graph of an acyclic Boolean conjunctive query. This motivates the following definition:

**Definition 2.6** Let $q$ be an acyclic Boolean conjunctive query. The attack graph of $q$ is the attack graph of $\tau$ for any join tree $\tau$ for $q$. We write $F \overset{q}{\rightsquigarrow} G$ (or simply $F \rightsquigarrow G$ if $q$ is clear from the context) to indicate that the attack graph of $q$ contains a directed edge from $F$ to $G$. ◁

### 2.2.3 Deciding the First-Order Definability

The main result of [Wijsen 2012] follows.

**Theorem 2.2** *Let $q$ be an acyclic* **SJFC** *query. The following statements are equivalent:*

1. *The attack graph of $q$ is acyclic.*
2. $\mathsf{CERTAINTY}(q)$ *is first-order expressible.*

Moreover, if the attack graph of $q$ is acyclic, we can effectively construct a first-order formula capturing $\mathsf{CERTAINTY}(q)$. We provide in Chapter 3 functions that construct such first-order formulas. The construction relies on the following definition [Wijsen 2012].

**Definition 2.7** Let $q$ be a Boolean conjunctive query without self-join. Let $R(\underline{\vec{x}}, \vec{y})$ be an atom of $q$, and let $\vec{y} = \langle y_1, y_2, \ldots, y_n \rangle$. Notice that $\vec{y}$ can contain constants and repeated variables. Let $\vec{z} = \langle z_1, z_2, \ldots, z_n \rangle$ be a sequence of distinct variables and let $C$ be a conjunction of equalities constructed as follows for $1 \leq i \leq n$,

1. If $y_i$ is a variable that does not occur in the sequence $\langle \vec{x}, y_1, y_2, \ldots, y_{i-1} \rangle$, then $z_i$ is identical to $y_i$;

2. otherwise, $z_i$ is a new variable and $C$ contains $z_i = y_i$.

Let $\vec{v}$ be a sequence of variables that contains exactly once each variable that occurs in $R(\underline{\vec{x}}, \vec{y})$. Let $\varphi(\vec{v})$ be a certain first-order rewriting for $q'(\vec{v})$, where $q'(\vec{v})$ is the nonBoolean conjunctive query whose set of atoms is $q \setminus \{R(\underline{\vec{x}}, \vec{y})\}$ (and whose free variables are $\vec{v}$). Obviously, if $q'$ is empty, then $\varphi = \mathbf{true}$. We define:

$$\mathit{Rewrite}(R(\underline{\vec{x}}, \vec{y}), q) = \exists \vec{v} \Big( R(\underline{\vec{x}}, \vec{y}) \wedge \forall \vec{z} \big( R(\underline{\vec{x}}, \vec{z}) \rightarrow (C \wedge \varphi(\vec{v})) \big) \Big).$$

If $q'(\vec{v})$ has no certain first-order rewriting, the value of $\mathit{Rewrite}(R(\underline{\vec{x}}, \vec{y}), q)$ is undefined. $\triangleleft$

Using the *Rewrite* function, one can compute a certain first-order rewriting. In particular, Lemma 8.6 and Lemma 8.10 in [Wijsen 2012] imply that if $F$ is a unattacked atom in the attack graph of $q$, if $\mathit{Rewrite}(F, q)$ is defined, then it is a certain first-order rewriting for $q$. The need to have an acyclic attack graph is then evident. If $q$ has an attack graph which is acyclic, then there is some unattacked atom $F$ in $q$. Let $q'(\vec{v})$ be the nonBoolean conjunctive query whose set of atom is $q \setminus \{F\}$ and whose free variables are $\vec{v}$. It was shown in [Wijsen 2012] that $q'(\vec{v})$ has an acyclic attack graph.

The construction of *Rewrite* can be roughly applied to compute a certain rewriting for queries that are not Boolean. The treatment was already mentioned in [Wijsen 2012]. The idea is to treat free variables as constants. Let $Q = (q, V)$ be a nonBoolean conjunctive query with $V = \{x_1, \ldots, x_n\}$. Let $c_1, \ldots, c_n$ be $n$ new constants. Let $q'$ be the query obtained from $q$ by replacing all occurrences of $x_i$ with $c_i$ ($1 \leq i \leq n$). Assume we have a first-order rewriting $\varphi$ for the Boolean conjunctive query $q'$. Then, a certain first-order rewriting for $Q$ can be obtained from $\varphi$ by replacing all occurrences of $c_i$ with $x_i$ ($1 \leq i \leq n$).

In Chapter 3, we will see how to use Definition 2.7 to effectively compute a certain first-order rewriting for acyclic (Boolean or nonBoolean) conjunctive queries whose attack graph is acyclic.

## 2.3 Certain FO Rewriting for Some Cyclic SJFC Queries

In the previous section, we showed that every acyclic **SJFC** query with an acyclic attack graph has a certain first-order rewriting. Theorem 2.3 identifies a class $\mathcal{C}$ of **SJFC** queries such that (1) every query in $\mathcal{C}$ has a certain first-order rewriting, (2) $\mathcal{C}$ contains $\mathcal{C}_{forest}$, and (3) $\mathcal{C}$ contains all acyclic **SJFC** queries with an acyclic attack graph.

**Definition 2.8** Let $q$ be an acyclic Boolean conjunctive query without self-join. A variable $x \in \mathsf{vars}(q)$ is called *top-reifiable in $q$* if for some unattacked atom $F$ of $q$, we have $x \in \mathsf{keyVars}(F)$. $\triangleleft$

**Theorem 2.3** *Let $q_1, \dots, q_n$ be acyclic Boolean conjunctive queries without self-join such that:*

1. *for all $i \in \{1, \dots, n\}$, the attack graph of $q_i$ is acyclic;*
2. *for all $i, j \in \{1, \dots, n\}$, if $i \neq j$, then $q_i$ and $q_j$ have no relation name in common; and*
3. *for all $i, j \in \{1, \dots, n\}$ such that $i \neq j$, if $x \in \mathsf{vars}(q_i) \cap \mathsf{vars}(q_j)$, then $x$ is top-reifiable in both $q_i$ and $q_j$.*

*Let $q$ be the Boolean conjunctive query defined as $q = \bigcup_{i=1}^{n} q_i$. Then, $q$ has no self-join and $\mathsf{CERTAINTY}(q)$ is first-order definable.*

Proof. Let $\vec{x} = \langle x_1, \dots, x_m \rangle$ be a sequence of variables that contains exactly once each variable that is shared among two distinct queries among $q_1, \dots, q_n$. Let $\vec{c} = \langle c_1, \dots, c_m \rangle$ be a vector of distinct constants not occurring in $q$. It can be easily seen that $q_{[\vec{x} \mapsto \vec{c}]}$ is an acyclic Boolean conjunctive query with an acyclic attack graph and thus $\mathsf{CERTAINTY}(q_{[\vec{x} \mapsto \vec{c}]})$ is first-order definable. We can assume a certain first-order rewriting $\varphi$ for $q_{[\vec{x} \mapsto \vec{c}]}$. Moreover, since $i \neq j$ implies that $q_{i[\vec{x} \mapsto \vec{c}]}$ and $q_{j[\vec{x} \mapsto \vec{c}]}$ have no variables or relation names in common, we can assume that $\varphi = \bigwedge_{i=1}^{n} \varphi_i$ where $\varphi_i$ is a certain first-order rewriting for $q_{i[\vec{x} \mapsto \vec{c}]}$.

For $1 \leq i \leq n$, let $\widehat{\varphi}_i(\vec{x})$ be the formula obtained from $\varphi_i$ by replacing each occurrence of $c_j$ with $x_j$ for $1 \leq j \leq m$. Let $\widehat{\varphi}(\vec{x}) = \bigwedge_{i=1}^{n} \widehat{\varphi}_i(\vec{x})$. Obviously, $\widehat{\varphi}(\vec{x})$ is a certain first-order rewriting for $q(\vec{x})$, where it is understood that the variables $x_1, \dots, x_m$ are free in $\widehat{\varphi}(\vec{x})$ and in $q(\vec{x})$. That is, for every uncertain database $\mathsf{db}$, for every $\vec{a} \in \mathbf{dom}^m$,

$$\mathsf{db} \models \widehat{\varphi}(\vec{a}) \iff \mathsf{db} \in \mathsf{CERTAINTY}(q(\vec{a})).$$

Consequently, for every database $\mathsf{db}$, if $\mathsf{db} \models \exists \vec{x} \widehat{\varphi}(\vec{x})$, then $\mathsf{db} \in \mathsf{CERTAINTY}(q)$.

We show next that the converse also holds, that is, for every database $\mathsf{db}$, if $\mathsf{db} \in \mathsf{CERTAINTY}(q)$, then $\mathsf{db} \models \exists \vec{x} \widehat{\varphi}(\vec{x})$. To this extent, let $\mathsf{db}$ be an uncertain database such that $\mathsf{db} \in \mathsf{CERTAINTY}(q)$. For $i \in \{1, \dots, n\}$, let $\mathsf{db}_i$ be the subset of $\mathsf{db}$ containing the facts whose relation name occurs in $q_i$. Thus, $\mathsf{db}$ is the disjoint union $\mathsf{db} = \mathsf{db}_1 \uplus \cdots \uplus \mathsf{db}_n$. The following property holds by the construction in the proof of Corollary 8.11 in [Wijsen 2012].

> For every $i \in \{1, \dots, n\}$, there exists a repair $\mathsf{rep}_i$ of $\mathsf{db}_i$ such that for every $\vec{a} \in \mathbf{dom}^m$, if $\mathsf{rep}_i$ satisfies $q_{i[\vec{x} \mapsto \vec{a}]}$, then every repair of $\mathsf{db}_i$ satisfies $q_{i[\vec{x} \mapsto \vec{a}]}$ (and hence $\mathsf{db}_i \models \widehat{\varphi}_i(\vec{a})$).

Let $\mathsf{rep} = \bigcup_{i=1}^{n} \mathsf{rep}_i$. Clearly, $\mathsf{rep}$ is a repair of $\mathsf{db}$. From $\mathsf{db} \in \mathsf{CERTAINTY}(q)$, it follows $\mathsf{rep} \models q$. We can assume $\vec{a}$ such that $\mathsf{rep} \models q_{[\vec{x} \mapsto \vec{a}]}$. For every $i \in \{1, \dots, n\}$, $\mathsf{rep}_i \models q_{i[\vec{x} \mapsto \vec{a}]}$, and hence $\mathsf{db}_i \models \widehat{\varphi}_i(\vec{a})$ by the aforementioned property. Clearly, $\mathsf{db} \models \widehat{\varphi}(\vec{a})$, hence $\mathsf{db} \models \exists \vec{x} \widehat{\varphi}(\vec{x})$. This concludes the proof. □

Notice that the query $q$ in Theorem 2.3 can be cyclic. We show in Theorem 2.4 that Theorem 2.3 generalizes the class $\mathcal{C}_{forest}$, whose definition follows. For example, query $\{R(\underline{x, y}, u),\ S(\underline{x, y}, u),\ T(\underline{x, z}),\ U(\underline{y, z})\}$ is covered by Theorem 2.3 but is not in $\mathcal{C}_{forest}$.

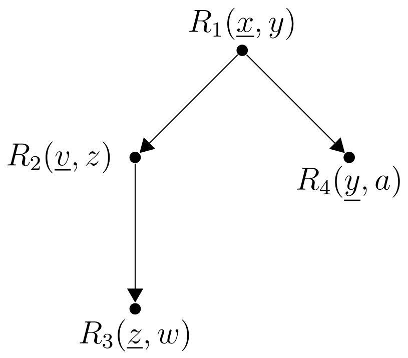

Figure 2.3: Fuxman-Miller join graph for Boolean query $q_3$ of Example 2.5.

**Definition 2.9** [Fuxman and Miller 2007] Let $q$ be a **SJFC** query. A Fuxman-Miller (FM) join graph $G$ of $q$ is a directed graph such that:

- the vertices of $G$ are the atoms of $q$;
- there is a directed edge from $F$ to $G$ if $F \neq G$ and there is some variable $y$ such that $y$ occurs at the position of a nonkey attribute in $F$ and $y$ occurs in $G$.

We say that $q \in \mathcal{C}_{forest}$ if

1. for every pair of distinct atoms $R, S$ in $q$, if a variable in $\mathsf{keyVars}(R)$ occurs at a nonkey position of $S$, then all the variables in $\mathsf{keyVars}(R)$ must occur at a nonkey position of $S$; and
2. the FM join graph of $q$ is a directed forest.

$\triangleleft$

**Example 2.5** The following conjunctive query is in the class $\mathcal{C}_{forest}$.

$$q_3 = \{R_1(\underline{x}, y), R_2(\underline{y}, z), R_3(\underline{z}, w), R_4(\underline{y}, a)\}.$$

The FM join graph of $q_3$ is shown in Figure 2.3. $\triangleleft$

**Theorem 2.4** *Let $q$ be a query in $\mathcal{C}_{forest}$. There exist $n \in \{1, \ldots, |q|\}$ and queries $q_1, \ldots, q_n$ with $q = \biguplus_{i=1}^{n} q_i$ such that $q_1, \ldots q_n$ satisfy the hypothesis of Theorem 2.3.*

Proof. Let $q$ be a query in $\mathcal{C}_{forest}$. Let $\tau$ be a FM join graph for $q$. Assume $\tau$ has $n$ connected components. Let $\tau_1, \ldots, \tau_n$ be the $n$ connected components of $\tau$. Let $q_i$ $(1 \leq i \leq n)$ be the query composed of the atoms of $\tau_i$. We show the three items of Theorem 2.3.

1. Every connected component of $\tau$ is in $\mathcal{C}_{tree}$. By Corollary 5 in [Wijsen 2009b], this implies $q_i$ $(1 \leq i \leq n)$ is acyclic and has an acyclic attack graph;
2. Obvious;

3. Assume $1 \leq i < j \leq n$ such that $x \in \mathsf{vars}(q_i) \cap \mathsf{vars}(q_j)$. We show that $x$ is top-reifiable both in $q_i$ and $q_j$. Let $R_i$ be the root of $\tau_i$ and $R_j$ be the root of $\tau_j$. By Lemma 2 in [Fuxman and Miller 2007], $x$ must occur in the key of $R_i$ and $R_j$. We show that $R_i$ is unattacked in the attack graph of $q_i$. The case $R_j$ is similar.

   By contradiction, assume there exists an atom $G$ in $q_i$ such that $G$ attacks $R_i$. Let $H \neq G$ be the last atom on the path from $R_i$ to $G$ in $\tau_i$. By Corollary 5 in [Wijsen 2009b], the FM join graph $\tau_i$ is a directed join tree for $q$ and by Lemma 4.9 in [Wijsen 2012], if $G$ attacks $R_i$ in $\tau_i$ then $G$ attacks every atom on the path between $G$ and $R_i$ in $\tau_i$, including $H$. Let $L = \mathsf{vars}(G) \cap \mathsf{vars}(H)$. As $G$ attacks $H$, we have $L \nsubseteq G^{+,q_i}$. As the FM join graph $\tau_i$ is acyclic, we know that the FM join graph $\tau_i$ contains no directed edge from $G$ to $H$. This implies that there is no variable in a nonkey position of $G$ that is in $\mathsf{vars}(H)$ and thus, the only shared variables between $G$ and $H$ must be in $\mathsf{keyVars}(G)$, this is $L \subseteq \mathsf{keyVars}(G)$. As $\mathsf{keyVars}(G) \subseteq G^{+,q_i}$, it must be the case that $L$ is a subset of $G^{+,q_i}$, a contradiction. We conclude that $x$ is top-reifiable in $R_i$.

$\square$

CHAPTER 3

# Certain Conjunctive Query Answering in First-Order Logic

The attack graph of an acyclic Boolean **SJFC** query $q$ turns out to be a key tool to decide the first-order definability of $\mathsf{CERTAINTY}(q)$. The attack graph of $q$ is acyclic if and only if $\mathsf{CERTAINTY}(q)$ is definable in first-order logic. In this chapter, we show how to define $\mathsf{CERTAINTY}(q)$ in first-order logic if the attack graph of $q$ is acyclic or, equivalently, how to provide a certain first-order rewriting for $q$.

In Section 3.1, we introduce a function that, given a (Boolean or nonBoolean) query $q$, computes a certain first-order rewriting $\varphi$ of $q$. We show that a direct translation of such rewriting $\varphi$ in SQL generally results in deeply nested SQL queries. This can lead to poor performance in database management systems.

We investigate in Section 3.2 how to syntactically simplify those SQL queries. We study the nesting and the number of quantifier blocks in certain first-order rewritings, and propose syntactic simplification techniques that aim to reduce these metrics. Such simplification techniques have an impact on the number of (nested) subqueries in the SQL queries and may decrease execution times.

This chapter is an extended version of the following scientific publication:

> "*Certain Conjunctive Query Answering in SQL*" in Scalable Uncertainty Management, volume 7530 of Lecture Notes in Computer Science, Springer Berlin Heidelberg [Decan et al. 2012].

## 3.1 Computation of Certain First-Order Rewritings

We provide on page 30 a function called `NaiveFo` that computes certain first-order rewritings. This function takes as input an acyclic self-join-free conjunctive query $Q = (q, V)$ whose attack graph is acyclic. The function implements Definition 2.7 and returns a certain first-order rewriting for $q$. Function `NaiveFo` handles both Boolean and nonBoolean queries. Importantly, `NaiveFo` makes use of function AttackGraph which, given an acyclic **SJFC** query $Q$, computes the attack graph of $Q$. The computation is effective and runs in quadratic time in the length of $Q$ [Wijsen 2012].

Notice that, although the construction of the attack graph is defined for Boolean queries, it is sufficient to treat free variables as constants to deal with nonBoolean queries. Let $Q = (q, V)$ be a nonBoolean conjunctive query and let $\vec{x} = \langle x_1, \ldots, x_n \rangle$ be a sequence of all the variables in $V$. Let $c_1, \ldots, c_n$ be $n$ new constants. The attack graph of $Q$ is obtained from the attack graph of $q_{[\vec{x} \mapsto \vec{c}]}$ by replacing all occurrences of each $c_i$ with $x_i$.

The following example illustrates a run of `NaiveFo`.

**Example 3.1** Consider the Boolean singleton query $q = \{R(\underline{x}, x, y, y, a)\}$. A call to `NaiveFo`$(q, \{\})$ eventually returns:

$$\varphi = \left[ \begin{array}{l} \exists x \exists y R(\underline{x}, x, y, y, a) \wedge \\ \forall z_1 \forall y \forall z_3 \forall z_4 \left[ R(\underline{x}, z_1, y, z_3, z_4) \rightarrow \left[ \begin{array}{l} z_1 = x \ \wedge \ z_3 = y \ \wedge \ z_4 = a \\ \wedge \ \textbf{true} \end{array} \right] \right] \end{array} \right]$$

There are two *foreach* loops in `NaiveFo`. The first *foreach* loop handles the variable that occurs in the primary key position of $R(\underline{x}, x, y, y, a)$ while the second *foreach* loop handles the nonkey positions. The set NEW is used to store the nonkey positions where a constant or a free variable occurs. Each position in the atom is considered once, and the first time a variable is treated by one of the loops, it is added to the set of free variables $V'$.

The call to `NaiveFo`$(q, \{\})$ chooses $F = R(\underline{x}, x, y, y, a)$. The first *foreach* loop ranges over the key positions of $R$ (since $k = 1$, the only value for $i$ is 1). The set X is populated with the nonfree variable $x$ and $x$ is added to the set $V'$ of free variables. Sets X and Y are used to store the variables that will be existentially quantified in the resulting formula. Set X stands for the variables occurring at key positions, while Y stands for the variables occurring at nonkey positions.

The second *foreach* loop ranges over the nonkey positions of $R$. It first considers $x$, then $y$ twice, and finally the constant $a$:

1. As $x$ is in the set $V'$, we create a new variable $z_1$ and we store in NEW the position 1. This new variable and this position are used together when computing $\varphi$: the expression $\bigwedge_{i \in \mathsf{NEW}} z_i = y_i$ will thus yield $z_1 = x$;

2. The first time $y$ is considered, $y$ is nonfree. The *else* statement is thus executed. We let $z_2$ be identical to $y$. This means that in the code that follows, one has to read $y$ whenever $z_2$ is used. Variable $y$ is then added to the set of free variables $V'$ and to the set Y of existentially quantified variables;

3. The second time $y$ is considered, it is now free. The *if* statement applies: $z_3$ is created and 3 is added to $\mathsf{NEW}$. This will result in the equality $z_3 = y$;

4. Finally, constant $a$ is considered. The *if* part is executed: variable $z_4$ is created and 4 is added to $\mathsf{NEW}$. This will add the equality $z_4 = a$.

After the second *foreach* loop, the function computes the resulting formula. The three lines after the loop are used to prepare the recursive call to `NaiveFo`. They compute a new query $q' = q \setminus \{F\}$ and restrict the set $V'$ of free variables to the variables that occur in $q'$.

A certain first-order rewriting $\varphi$ is then computed:

1. For each variable in $\mathsf{X}$ and $\mathsf{Y}$, an existential quantifier is used. In our example, this leads to $\exists x \exists y\ R(\underline{x}, x, y, y, a)$;

2. For each variable $z_i$, a universal quantifier is used. As there is no need to introduce a new variable $z_2$, we reuse $y$ instead (see second item above).

   This gives $\forall z_1 \forall y \forall z_3 \forall z_4\ R(\underline{x}, z_1, y, z_3, z_4)$.

3. The right part of the implication is composed of two parts: the first part is a conjunction of equalities: each position stored in $\mathsf{NEW}$ is used to associate the new variables $z_i$ to some variable or constant in the atom. In this example, the generated equalities are $z_1 = x$, $z_3 = y$ and $z_4 = a$. The second part consists of the recursive call. In our case, this call will return **true** as $q' = \{\}$.

$\lhd$

This function is called "naive" because it just implements Definition 2.7. As each call to `NaiveFo` produces a *block* of existential quantifiers and a *block* of universal quantifiers, the output of this function may lead to a high quantifier rank or to a high number of quantifier alternations. This may be problematic when considering the execution of such queries in a database management system. A direct translation of such rewriting in SQL generally results in several subqueries and in deeply nested SQL queries. Subqueries and nested subqueries in SQL queries can lead to poor performance in database management systems. The SQL query of Example 3.2 contains only two nested subqueries and already has poor performance.

**Example 3.2** Consider schema $S$ with 2 relations, $P$ and $D$, each of signature $[2, 2]$. Fact $P(p, s)$ means that patient $p$ has symptom $s$. Fact $D(d, s)$ means that $s$ is a symptom of disease $d$. The following query gets pairs $(p, d)$ such that patient $p$ has all symptoms of disease $d$.

$$\psi_{\text{dis}}(p, d) = \exists s \Big( P(\underline{p, s}) \land D(\underline{d, s}) \land \forall s' \big( D(\underline{d, s'}) \to P(\underline{p, s'}) \big) \Big).$$

An SQL translation of $\psi_{\text{dis}}(p, d)$ follows. This SQL query contains two nested `NOT EXISTS` subqueries which are related to the universal quantifier in $\psi_{\text{dis}}(p, d)$.

**Function** NaiveFo($q$,$V$) constructs certain first-order rewriting.

```
Input: $Q = (q, V)$ is an acyclic SJFC query whose attack graph is acyclic and
       where $V$ is the set of free variables of $q$.
Result: certain first-order rewriting $\varphi$ for $Q$.
begin
    if $q = \emptyset$ then
        $\varphi \leftarrow$ true;
    else
        $E \leftarrow \mathsf{AttackGraph}((q, V))$;
        choose $F = R(\underline{x_1, \dots, x_k}, y_1, \dots, y_\ell)$ in $q$ such that $\forall G \in q : (G, F) \notin E$;
        $V' \leftarrow V$;
        $\mathsf{X} \leftarrow \emptyset$;
        foreach $i \leftarrow 1$ to $k$ do
            if $x_i$ is a variable and $x_i \notin V'$ then
                $V' \leftarrow V' \cup \{x_i\}$;
                $\mathsf{X} \leftarrow \mathsf{X} \cup \{x_i\}$;
        $\mathsf{Y} \leftarrow \emptyset$;
        $\mathsf{NEW} \leftarrow \emptyset$;
        foreach $i \leftarrow 1$ to $\ell$ do
            if $y_i$ is a constant or $y_i \in V'$ then
                let $z_i$ be a new variable;
                $\mathsf{NEW} \leftarrow \mathsf{NEW} \cup \{i\}$;
            else                                   /* $y_i$ is a variable not in $V'$ */
                let $z_i$ be the same variable as $y_i$;
                $V' \leftarrow V' \cup \{y_i\}$;
                $\mathsf{Y} \leftarrow \mathsf{Y} \cup \{y_i\}$;
        $q' \leftarrow q \setminus \{F\}$;
        $V' \leftarrow V' \cap \mathsf{vars}(q')$;
        $\varphi \leftarrow \left[\begin{array}{l} \exists \mathsf{X}\, \exists \mathsf{Y} R(\underline{x_1, \dots, x_k}, y_1, \dots, y_\ell) \wedge \\ \forall z_1 \dots \forall z_\ell \left[ R(\underline{x_1, \dots, x_k}, z_1, \dots, z_\ell) \rightarrow \left[\begin{array}{l} \bigwedge_{i \in \mathsf{NEW}} z_i = y_i \\ \wedge\ \mathtt{NaiveFo}(q', V') \end{array}\right]\right]\end{array}\right]$;
    return $\varphi$;
```

```
PSI_DIS = SELECT p1.PAT, d1.DIS
          FROM   P AS p1, D AS d1
          WHERE  NOT EXISTS ( SELECT *
                              FROM   D AS d2
                              WHERE  d2.DIS = d1.DIS
                              AND    NOT EXISTS ( SELECT *
                                                  FROM   P AS p2
                                                  WHERE  p2.PAT = p1.PAT
                                                  AND    p2.SYM = d2.SYM ) )
```

Although PSI_DIS only contains two nested subqueries, it already has poor performance: it takes more than 30 seconds to get the answer to this query on a consistent database of $10,000$ tuples[1]. $\triangleleft$

As NaiveFo produces certain first-order rewritings $\varphi$ with several alternation of (blocks of) $\exists$ and $\forall$, the translations in SQL of $\varphi$ can contain several nested NOT EXISTS subqueries and can have high execution times, even for conjunctive queries with few atoms. We will show in the next two sections how to syntactically simplify the certain first-order rewritings in order to reduce the number of (nested) subqueries in their SQL translations.

**Example 3.3** Let $q$ be the query $\{R_1(\underline{x_1}, x_2), R_2(\underline{x_2}, x_3), R_3(\underline{x_3}, a)\}$. A certain first-order rewriting of $q$ is given by the following formula $\varphi$.

$$\begin{aligned}\varphi = \exists x_1 \exists x_2 R_1(\underline{x_1}, x_2) \wedge \forall x_2' R_1(\underline{x_1}, x_2') \rightarrow \\ \Big(\exists x_3 R_2(\underline{x_2'}, x_3) \wedge \forall x_3' R_2(\underline{x_2'}, x_3') \rightarrow \\ \big(R_3(\underline{x_3'}, a) \wedge \forall z R_3(\underline{x_3'}, z) \rightarrow z = a\big)\Big)\end{aligned}$$

Assume the attributes of $R_i$ are $\langle A, B\rangle$ ($1 \leq i \leq 3$). The following query is a translation of $\varphi$ in SQL.

```
SELECT 'true' FROM R1 as r1 WHERE NOT EXISTS
  (SELECT * FROM R1 as r1p WHERE r1p.A = r1.A AND NOT EXISTS
    (SELECT * FROM R2 as r2 WHERE r2.A = r1p.B AND NOT EXISTS
      (SELECT * FROM R2 as r2p WHERE r2p.A = r2.A AND NOT EXISTS
        (SELECT * FROM R3 as r3 WHERE r3.A = r2p.B AND r3.B = 'a' AND NOT EXISTS
          (SELECT * FROM R3 as r3p WHERE r3p.A = r3.A AND r3p.B <> 'a')))))
```

Although $q$ is composed of only 3 atoms and 3 variables, the SQL translation of $\varphi$ already contains 5 nested NOT EXISTS. $\triangleleft$

In the following example, we give a query and three certain first-order rewritings for this query. Those rewritings are syntactically very different and also result in very different SQL queries.

[1]The experimental setup will be explained in Section 4.2.

```
Q3 = SELECT 'TRUE' FROM R1 r11, R2 r21, R3 r31
     WHERE r11.B = 'b' AND r21.B = 'b' AND r31.B = 'b';

N3 = SELECT 'TRUE' FROM R1 r11
     WHERE r11.B = 'b' AND NOT EXISTS (
        SELECT * FROM R1 r12
        WHERE r12.A = r11.A AND ( r12.B <> 'b' OR NOT EXISTS (
          SELECT * FROM R2 r21
          WHERE r21.B = 'b' AND NOT EXISTS (
            SELECT * FROM R2 r22
            WHERE r22.A = r21.A AND ( r22.B <> 'b' OR NOT EXISTS (
              SELECT * FROM R3 r31
              WHERE r31.B = 'b' AND NOT EXISTS (
                SELECT * FROM R3 r32
                WHERE r32.A = r31.A AND r32.B <> 'b')))))));

R3 = SELECT 'TRUE' FROM R1 r11 WHERE r11.B = 'b' AND NOT EXISTS
        (SELECT * FROM R1 r12 WHERE r12.A = r11.A AND r12.B <> 'b')
     INTERSECT SELECT 'TRUE' FROM R2 r21 WHERE r21.B = 'b' AND NOT EXISTS
        (SELECT * FROM R2 r22 WHERE r22.A = r21.A AND r22.B <> 'b')
     INTERSECT SELECT 'TRUE' FROM R3 r31 WHERE r31.B = 'b' AND NOT EXISTS
        (SELECT * FROM R3 r32 WHERE r32.A = r31.A AND r32.B <> 'b');

B3 = SELECT 'TRUE' FROM R1 r11, R2 r21, R3 r31
     WHERE r11.B = 'b' AND r21.B = 'b' AND r31.B = 'b' AND NOT EXISTS (
        SELECT * FROM R1 r12, R2 r22, R3 r32
        WHERE r11.A = r12.A AND r21.A = r22.A AND r32.A = r31.A AND
          (r12.B <> 'b' OR r22.B <> 'b' OR r32.B <> 'b'));
```

Figure 3.1: SQL translations `Q3`, `N3`, `R3` and `B3` of Examples 3.4, 3.11 and 3.14.

**Example 3.4** For each $m \geq 1$, assume relation name $R_i$ with signature $[2,1]$, and let $\lfloor m \rfloor = \{R_1(\underline{x_1}, b), \ldots, R_m(\underline{x_m}, b)\}$, where $b$ is a constant. For $m \geq 1$, let $\llbracket m \rrbracket = (\lfloor m \rfloor, \emptyset)$. Thus $\llbracket m \rrbracket$ represents the first-order sentence $\exists x_1 \ldots \exists x_m \big(R_1(\underline{x_1}, b) \wedge \cdots \wedge R_m(\underline{x_m}, b)\big)$, a Boolean query whose attack graph has no edge. Formulas $\psi_1$, $\psi_2$, and $\psi_3$ are three possible certain first-order rewritings for $\llbracket m \rrbracket$. The formula $\psi_1$ is returned by function `NaiveFo`, while $\psi_2$ and $\psi_3$ result from some syntactic simplification techniques described in Subsections 3.2.2 and 3.2.3. In particular, $\psi_2$ minimizes the nesting depth of quantifier blocks, and $\psi_3$ minimizes the number of quantifier blocks.

$$\begin{aligned}
\psi_1 \;=\; & \exists x_1 \Bigg( R_1(\underline{x_1}, b) \wedge \forall z_1 \Bigg( R_1(\underline{x_1}, z_1) \rightarrow z_1 = b \wedge \\
& \quad \exists x_2 \Big( R_2(\underline{x_2}, b) \wedge \forall z_2 \Big( R_2(\underline{x_2}, z_2) \rightarrow z_2 = b \wedge \\
& \quad\quad \ddots \\
& \quad\quad\quad \exists x_m \big( R_m(\underline{x_m}, b) \wedge \forall z_m \big( R_m(\underline{x_m}, z_m) \rightarrow z_m = b \big)\big) \ldots \Big)\Big)\Bigg)\Bigg)
\end{aligned}$$

$$\psi_2 \;=\; \bigwedge_{i=1}^{m} \exists x_i \Big( R_i(\underline{x_i}, b) \wedge \forall z_i \big( R_i(\underline{x_i}, z_i) \rightarrow z_i = b \big)\Big)$$

$$\psi_3 \;=\; \exists x_1 \ldots \exists x_m \Big( \bigwedge_{i=1}^{m} R_i(\underline{x_i}, b) \wedge \forall z_1 \ldots \forall z_m \big( \bigwedge_{i=1}^{m} R_i(\underline{x_i}, z_i) \rightarrow \bigwedge_{i=1}^{m} z_i = b \big)\Big)$$

Notice that $\psi_1$, $\psi_2$, and $\psi_3$ are semantically equivalent and each contain $m$ existential and $m$ universal quantifiers. The differences in syntactic complexity persist in SQL. Assume that, for each $i \in \{1, \ldots, m\}$, the first and the second attribute of each relation $R_i$ are respectively named $A$ and $B$. Thus, $A$ is the primary key attribute.

For $m = 3$, the resulting SQL queries are shown in Figure 3.1. `Q3` is the conjunctive query while `N3`, `R3` and `B3` are direct translations into SQL of $\psi_1$, $\psi_2$, and $\psi_3$.

The fact that $\psi_3$ only has one $\forall$ quantifier block results in `B3` having only one `NOT EXISTS`. Notice further that `B3` requires $m$ tables in each `FROM` clause, whereas `R3` takes the intersection of $m$ SQL queries, each with a single table in the `FROM` clause. $\lhd$

This example illustrates the possibility to consider syntactic optimizations in order to reduce the number of (nested) subqueries in the SQL translations of the certain first-order rewritings.

## 3.2 Syntactic Simplifications

We introduce three metrics for first-order queries: the *quantifier rank*, the *quantifier block rank* and the *number of quantifier blocks*. We see how those metrics are related to the number of (nested) subqueries in the SQL translations and we propose two syntactic optimizations that aim to reduce those metrics in certain first-order rewritings. After that, we will tackle an important practical question: do these optimizations result in faster SQL queries on real-life databases? We answer this question in Chapter 4.

### 3.2.1 Preliminaries

The *quantifier rank* of a first-order formula $\varphi$, denoted by $\mathsf{qr}(\varphi)$, is the depth of the quantifier nesting in $\varphi$ and is defined as usual (see, for example, [Libkin 2004, page 32]):

- If $\varphi$ is quantifier-free, then $\mathsf{qr}(\varphi) = 0$.
- $\mathsf{qr}(\varphi_1 \wedge \varphi_2) = \mathsf{qr}(\varphi_1 \vee \varphi_2) = \max\big(\mathsf{qr}(\varphi_1), \mathsf{qr}(\varphi_2)\big)$;
- $\mathsf{qr}(\neg\varphi) = \mathsf{qr}(\varphi)$;
- $\mathsf{qr}(\exists x\varphi) = \mathsf{qr}(\forall x\varphi) = 1 + \mathsf{qr}(\varphi)$.

The quantifier rank of certain first-order queries was studied in [Decan et al. 2012]. The quantifier rank may reflect the depth of nested subqueries in SQL. However, it is not true that every quantifier leads to an additional subquery in SQL. Quantifiers of the same type that are regrouped usually result in only one additional subquery in SQL. This is illustrated in the next example.

**Example 3.5** Consider the following conjunctive queries $q_1$ and $q_2$.

$$q_1 = \exists x\Big(R(\underline{x}, b) \wedge \forall y\big(R(\underline{x}, y) \rightarrow y = b\big)\Big)$$
$$q_2 = \exists x\exists u\exists v\exists w\Big(S(\underline{x}, b, u, v, w) \wedge \forall y\forall u'\forall v'\forall w'\big(S(\underline{x}, y, u', v', w') \rightarrow y = b\big)\Big)$$

The quantifier rank of $q_1$ is 2 while the quantifier rank of $q_2$ is 8. Their respective SQL translations follow. Despite having different quantifier ranks, the two first-order queries result in SQL translations having the same number of subqueries. The additional quantifiers in $q_2$ do not lead to additional subqueries in its SQL translation.

```
Q1 = SELECT 'true' FROM R as r1 WHERE r1.B = 'b' AND NOT EXISTS
       (SELECT * FROM R as r2 WHERE r2.A = r1.A AND r2.B <> 'b')
Q2 = SELECT 'true' FROM S as s1 WHERE s1.B = 'b' and NOT EXISTS
       (SELECT * FROM S as s2 WHERE s2.A = s1.A AND s2.B <> 'b')
```

$\triangleleft$

We now define the quantifier block rank of a query as a new metric based on the quantifier rank. In contrast with the quantifier rank, the quantifier block rank does not take into account successive quantifiers of the same type.

A first-order formula $\varphi$ is said to be in *prenex normal form* if it is of the form $Q_1x_1 \ldots Q_nx_n\psi$, where $Q_i$'s are either $\exists$ or $\forall$ and $\psi$ is quantifier-free. We say that $\varphi$ has a *quantifier block rank* $m$ if $Q_1x_1 \ldots Q_nx_n$ can be divided into $m$ blocks such that all quantifiers in a block are of the same type and quantifiers in two consecutive blocks are different. For queries that are not in prenex normal form, the quantifier block rank is computed as follows.

**Definition 3.1** A universally quantified formula is a formula whose top-most connective in the syntax tree is $\forall$. An existentially quantified formula is a formula whose top-most connective in the syntax tree is $\exists$. The *quantifier block rank* of a first-order formula $\varphi$, denoted $\mathsf{qbr}(\varphi)$, is defined as follows.

- If $\varphi$ is quantifier-free, then $\mathsf{qbr}(\varphi) = 0$.
- $\mathsf{qbr}(\varphi_1 \wedge \varphi_2) = \mathsf{qbr}(\varphi_1 \vee \varphi_2) = \max\big(\mathsf{qbr}(\varphi_1), \mathsf{qbr}(\varphi_2)\big)$;
- $\mathsf{qbr}(\neg\varphi) = \mathsf{qbr}(\varphi)$;
- if $\varphi$ is not universally quantified and $n \geq 1$, then $\mathsf{qbr}(\forall x_1 \dots \forall x_n \varphi) = 1 + \mathsf{qbr}(\varphi)$;
- if $\varphi$ is not existentially quantified and $n \geq 1$, then $\mathsf{qbr}(\exists x_1 \dots \exists x_n \varphi) = 1 + \mathsf{qbr}(\varphi)$.

$\lhd$

Consider the queries $q_1$ and $q_2$ of Example 3.5. Those queries have a quantifier block rank equal to 2. We will see in other examples that the quantifier block rank reflects the depth of nested subqueries in SQL.

We introduce the *quantifier block number* of a formula $\varphi$, denoted $\mathsf{qbn}(\varphi)$, as the total number of quantifier blocks in $\varphi$.

**Example 3.6** Let $\varphi$ be $\exists x \exists y(\exists u \varphi_1 \wedge \exists v \varphi_2)$ where $\varphi_1$, $\varphi_2$ are both quantifier-free, then $\mathsf{qbn}(\varphi) = 3$ and $\mathsf{qbr}(\varphi) = 2$. Notice that $\exists u \varphi_1 \wedge \exists v \varphi_2$ is not an existentially quantified formula, because its top-most connective in the syntax tree is $\wedge$. $\lhd$

Clearly, if $\varphi$ is in prenex normal form, then the quantifier block rank of $\varphi$ is equal to its number of quantifier blocks, i.e. $\mathsf{qbr}(\varphi) = \mathsf{qbn}(\varphi)$. For formulas which are not in prenex normal form, the number of quantifier blocks is counted as follows.

**Definition 3.2** Let $\varphi$ be a first-order formula. The *number of quantifier blocks* in $\varphi$, denoted by $\mathsf{qbn}(\varphi)$ is:

- If $\varphi$ is quantifier-free, then $\mathsf{qbn}(\varphi) = 0$.
- $\mathsf{qbn}(\varphi_1 \wedge \varphi_2) = \mathsf{qbn}(\varphi_1 \vee \varphi_2) = \mathsf{qbn}(\varphi_1) + \mathsf{qbn}(\varphi_2)$;
- $\mathsf{qbn}(\neg\varphi) = \mathsf{qbn}(\varphi)$;
- if $\varphi$ is not universally quantified and $n \geq 1$, then $\mathsf{qbn}(\forall x_1 \dots \forall x_n \varphi) = 1 + \mathsf{qbn}(\varphi)$;
- if $\varphi$ is not existentially quantified and $n \geq 1$, then $\mathsf{qbn}(\exists x_1 \dots \exists x_n \varphi) = 1 + \mathsf{qbn}(\varphi)$.

$\lhd$

Every first-order formula has an equivalent one in prenex normal form with a lower or equal number of quantifier blocks.

**Example 3.7** Let $\varphi$ be the formula of Example 3.6. Let $\psi = \exists x \exists y \exists u \exists v(\varphi_1 \wedge \varphi_2)$ be an equivalent formula in prenex normal form. We have $\mathsf{qbn}(\psi) = 1$ and $\mathsf{qbn}(\varphi) = 3$. $\lhd$

**Proposition 3.1** *For every first-order formula $\varphi$, there exists a first-order formula $\psi$ such that $\psi \equiv \varphi$, $\psi$ is in prenex normal form and $\mathsf{qbn}(\psi) \leq \mathsf{qbn}(\varphi)$.*

Proof. The proof runs by induction on the structure of $\varphi$. The result is obvious if $\varphi$ is quantifier free. For the induction step, we distinguish the following cases:

- Case $\varphi = \varphi_1 \wedge \varphi_2$ or $\varphi = \varphi_1 \vee \varphi_2$. Assume $\mathsf{qbn}(\varphi_1) = n_1$ and $\mathsf{qbn}(\varphi_2) = n_2$. By the induction hypothesis, we can assume integers $m_1 \leq n_1$ and $m_2 \leq n_2$ such that $\varphi_1$ has an equivalent formula $\varphi_1'$ in prenex normal form with a number of quantifier blocks $m_1$, and $\varphi_2$ has an equivalent formula $\varphi_2'$ in prenex normal form with a number of quantifier blocks $m_2$. A standard translation [Papadimitriou 1994, page 99] of $\varphi_1' \wedge \varphi_2'$ in prenex normal form yields a formula with a number of quantifier blocks $\leq m_1 + m_2$. Finally, notice $m_1 + m_2 \leq n_1 + n_2 = \mathsf{qbn}(\varphi)$.

- Case $\varphi = \neg\varphi_1$. Easy.

- Case $\varphi = \forall x_1 \ldots \forall x_n \varphi_1$ with $n \geq 1$ and $\varphi_1$ is not universally quantified. Assume $\mathsf{qbn}(\varphi_1) = n_1$. By the induction hypothesis, we can assume integer $m_1 \leq n_1$ such that $\varphi_1$ has an equivalent formula $\varphi_1'$ in prenex normal form with a number of quantifier blocks $m_1$. Obviously, $\forall x_1 \ldots \forall x_n \varphi_1'$ is equivalent to $\varphi$, is in prenex normal form, and has a number of quantifier blocks $\leq 1 + m_1$. Finally, notice $1 + m_1 \leq 1 + n_1 = \mathsf{qbn}(\varphi)$.

- Case $\varphi = \exists x_1 \ldots \exists x_n \varphi_1$ with $n \geq 1$ and $\varphi_1$ is not existentially quantified. Analogous to the previous case.

□

Formulas returned by function `NaiveFo` can be "optimized" so as to have lower quantifier (block) rank and/or less quantifier blocks. Consider the penultimate line of function `NaiveFo`, which specifies the first-order formula $\varphi$ returned by a call `NaiveFo`$(q, V)$ with $q \neq \emptyset$. Since `NaiveFo` is called recursively once for each atom of $q$, the algorithm can return a formula with $2|q|$ quantifier blocks and with a quantifier block rank as high as $2|q|$.

In the following subsections, we present some theoretical results that can be used to construct "improved" or "simpler" certain first-order rewritings. Subsection 3.2.2 presents results to decrease the quantifier block rank. In Subsection 3.2.3, we show that `NaiveFo` can be easily modified so as to return formulas with less (alternations of) quantifier blocks. Importantly, our simplifications do not decrease (nor increase) the total number of quantifiers in a formula; they merely group quantifiers of the same type in blocks and/or decrease the nesting depth of quantifiers.

### 3.2.2 Reduction of the Quantifier (Block) Rank

Consider query $\llbracket m \rrbracket$ with $m \geq 1$ in Example 3.4. Function `NaiveFo` will "rewrite" the atoms $R_i(\underline{x_i}, b)$ sequentially ($1 \leq i \leq m$). However, since these atoms have no bound variables in common, it is correct to rewrite them "in parallel" and then join the resulting formulas.

**Example 3.8** Let $q_3 = \exists x_1 \exists x_2 \exists x_3 \big(R_1(\underline{x_1}, b) \wedge R_2(\underline{x_2}, b) \wedge R_3(\underline{x_3}, b)\big)$. This is, $q_3 = \lfloor 3 \rfloor$. A naive certain first-order rewriting of $q_3$ is the following sentence $\varphi_3$:

$$\varphi_3 = \begin{array}{l} \exists x_1 \Bigg[ R_1(\underline{x_1}, b) \wedge \forall y_1 \Bigg( R_1(\underline{x_1}, y_1) \rightarrow y_1 = b \wedge \\ \quad \exists x_2 \Bigg[ R_2(\underline{x_2}, b) \wedge \forall y_2 \Bigg( R_2(\underline{x_2}, y_2) \rightarrow y_2 = b \wedge \\ \quad\quad \exists x_3 \big[ R_3(\underline{x_3}, b) \wedge \forall y_3 \big( R_3(\underline{x_3}, y_3) \rightarrow y_3 = b \big) \big] \Bigg) \Bigg] \Bigg) \Bigg] \end{array}$$

An "improved" certain first-order rewriting is given by the following query $\rho_3$. As atoms $R_1(\underline{x_1}, b)$, $R_2(\underline{x_2}, b)$ and $R_3(\underline{x_3}, b)$ have no common variable, certain first-order rewritings are computed for each of them and then joined together.

$$\rho_3 = \begin{array}{l} \exists x_1 \Big( R_1(\underline{x_1}, b) \wedge \forall y_1 \big( R_1(\underline{x_1}, y_1) \rightarrow y_1 = b \big) \Big) \wedge \\ \exists x_2 \Big( R_2(\underline{x_2}, b) \wedge \forall y_2 \big( R_2(\underline{x_2}, y_2) \rightarrow y_2 = b \big) \Big) \wedge \\ \exists x_3 \Big( R_3(\underline{x_3}, b) \wedge \forall y_3 \big( R_3(\underline{x_3}, y_3) \rightarrow y_3 = b \big) \Big) \end{array}$$

Queries $\varphi_3$ and $\rho_3$ contain exactly the same number of quantifiers of each type. Those quantifiers are nested in $\varphi_3$ while they are spread in three "independent" parts in $\rho_3$. $\triangleleft$

The idea is generalized in Theorem 3.1.

**Definition 3.3** Let $Q = (q, V)$ be an **SJFC** query with $q \neq \emptyset$. An *independent partition* of $Q$ is a (complete disjoint) partition $\{q_1, \ldots, q_k\}$ of $q$ such that for $1 \leq i < j \leq k$, $\mathsf{vars}(q_i) \cap \mathsf{vars}(q_j) \subseteq V$. $\triangleleft$

**Theorem 3.1** *Let $Q = (q, V)$ be an acyclic* **SJFC** *query. Let $\{q_1, \ldots, q_k\}$ be an independent partition of $Q$. For each $1 \leq i \leq k$, let $\varphi_i$ be a certain first-order rewriting for $Q_i = (q_i, V_i)$, where $V_i = V \cap \mathsf{vars}(q_i)$. Then, $\bigwedge_{i=1}^{k} \varphi_i$ is a certain first-order rewriting for $Q$.*

Proof. Let $\theta$ be an arbitrary valuation over $V$. Let $\vec{x}$ be the ordered sequence of variables in $V$. For $1 \leq i \leq k$, let $\theta_i$ be the restriction of $\theta$ on $V_i$, and let $\vec{x}_i$ be the ordered sequence of variables in $V_i$. Define $\varphi = \bigwedge_{i=1}^{k} \varphi_i$.

Let $\mathsf{db}$ be a database such that $\mathsf{db} \models \varphi(\theta(\vec{x}))$. Then, for $1 \leq i \leq k$, $\mathsf{db} \models \varphi_i(\theta_i(\vec{x}_i))$. Let $\mathsf{rep}$ be an arbitrary repair of $\mathsf{db}$. Since each $\varphi_i$ is a certain first-order rewriting for $Q_i$, we have that for $1 \leq i \leq k$, $\theta_i(\vec{x}_i) \in Q_{i\mathsf{sure}}(\mathsf{db})$, hence $\theta_i(\vec{x}_i) \in Q_i(\mathsf{rep})$. So, for every $1 \leq i \leq k$, we can extend $\theta_i$ to a valuation $\Theta_i$ over $\mathsf{vars}(q_i)$ such that $\Theta_i(q_i) \subseteq \mathsf{rep}$. Let $\Theta = \bigcup_{i=1}^{k} \Theta_i$. We show that $\Theta$ is a function. Assume that the same variable $x$ occurs in $\mathsf{vars}(q_i)$ and $\mathsf{vars}(q_j)$ with $i \neq j$. Since $\{q_1, \ldots, q_k\}$ is an independent partition of $Q$, we have $x \in V$, consequently $x \in V_i$ and $x \in V_j$. It follows that $\Theta_i(x) = \theta(x)$ and $\Theta_j(x) = \theta(x)$, hence $\Theta_i(x) = \Theta_j(x)$. Since $\Theta(q) \subseteq \mathsf{rep}$ is now obvious, we have $\Theta(\vec{x}) = \theta(\vec{x}) \in Q(\mathsf{rep})$. Since $\mathsf{rep}$ is an arbitrary repair of $\mathsf{db}$, it follows $\theta(\vec{x}) \in Q_{\mathsf{sure}}(\mathsf{db})$.

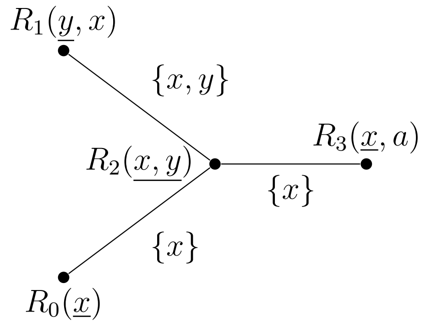

Figure 3.2: A join tree $\tau$ for query $q$ of Example 3.9.

Conversely, assume $\theta(\vec{x}) \in Q_{\mathsf{sure}}(\mathsf{db})$. Let $\mathsf{rep}$ be a repair of $\mathsf{db}$. We have $\theta(\vec{x}) \in Q(\mathsf{rep})$. So, we can extend $\theta$ to a valuation $\Theta$ over $\mathsf{vars}(q)$ such that $\Theta(q) \subseteq \mathsf{rep}$. For $1 \leq i \leq k$, let $\Theta_i$ be the restriction of $\Theta$ on $\mathsf{vars}(q_i)$. Then for $1 \leq i \leq k$, $\Theta_i(q_i) \subseteq \mathsf{rep}$. It follows $\Theta_i(\vec{x}_i) \in Q_i(\mathsf{rep})$. As $\Theta(\vec{x}_i) = \theta_i(\vec{x}_i)$ is obvious and since $\mathsf{rep}$ is an arbitrary repair of $\mathsf{db}$, it follows that for $1 \leq i \leq k$, $\theta_i(\vec{x}_i) \in Q_{i\mathsf{sure}}(\mathsf{db})$. As each $\varphi_i$ is a certain first-order rewriting for $Q_i$, we have that for $1 \leq i \leq k$, $\mathsf{db} \models \varphi_i(\theta_i(\vec{x}_i))$. Consequently, $\mathsf{db} \models \bigwedge_{i=1}^{k} \varphi_i(\theta_i(\vec{x}_i))$. It follows $\mathsf{db} \models \varphi(\theta(\vec{x}))$. □

Intuitively, given a join tree $\tau$, we define $\mathsf{diameter}(\tau)$ as the maximal sum of arities found on any path in $\tau$. An example follows.

**Example 3.9** Let $q = \{R_0(\underline{x}), R_1(\underline{y}, x), R_2(\underline{x, y}), R_3(\underline{x}, a)\}$, an acyclic **SJFC** query. Figure 3.2 shows a join tree $\tau$ for $q$. The maximal sum of arities is given by the chain $\{R_1(\underline{y}, x), R_2(\underline{x, y}), R_3(\underline{x}, a)\}$. This join tree has $\mathsf{diameter}(\tau) = 6$.

◁

**Definition 3.4** Let $Q = (q, V)$ be an acyclic **SJFC** query. Let $\tau$ be a join tree for $Q$. A *chain* in $\tau$ is a subset $q' \subseteq q$ such that the subgraph of $\tau$ induced by $q'$ is a path graph. We define $\mathsf{diameter}(\tau)$ as the largest integer $n$ such that $n = \Sigma_{F \in q'} \mathsf{arity}(F)$ for some chain $q'$ in $\tau$. ◁

**Corollary 3.1** *Let $Q = (q, V)$ be an acyclic* **SJFC** *query whose attack graph is acyclic. Let $\tau$ be a join tree for $Q$. There exists a certain first-order rewriting $\varphi$ for $Q$ such that $\mathsf{qr}(\varphi) \leq \mathsf{diameter}(\tau)$.*

Proof. The proof makes use of the following sublemma.

**Sublemma 3.1** *[Wijsen 2012, Lemma C.1] Let $Q = (q, V)$ be an acyclic conjunctive query without self-join with an acyclic attack graph. Let $x$ be a variable that occurs in $q$. Let $Q' = (q, V \cup \{x\})$ then, $Q'$ is an acyclic conjunctive query without self-join with an acyclic attack graph. Moreover, for $F \in Q$, if $F$ is unattacked in $Q$, then $F$ is unattacked in $Q'$.*

The proof of Corollary 3.1 runs by induction on $|q|$. The desired result holds obviously for $|q| = 0$. For the induction step, assume $|q| \geq 1$. We can assume an unattacked atom

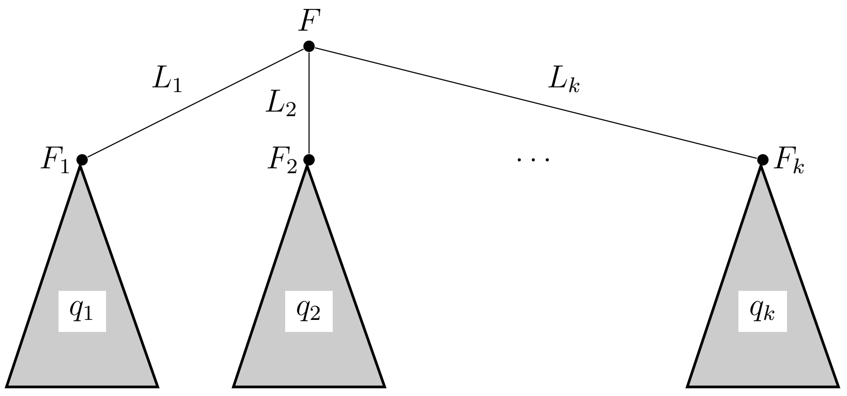

Figure 3.3: Join tree $\tau_F$.

$F = R(\underline{\vec{x}}, \vec{y})$ in $Q$, and let $\ell$ be the arity of $R$. Let $Q' = (q', V')$ where $q' = q \setminus \{F\}$ and $V' = \big(V \cup \mathsf{vars}(F)\big) \cap \mathsf{vars}(q')$.

Let $\tau$ be a join tree for $Q$, and denote by $\tau_F$ the directed rooted join tree obtained from $\tau$ by selecting $F$ as the root. This situation is depicted in Figure 3.3. Let $F_1, F_2, \dots, F_k$ be the children of $F$ in $\tau_F$. For $1 \leq i \leq k$, let $q_i$ be the set of atoms that contains $F_i$ and all its descendants, and let $\tau_i$ be the subtree of $\tau$ induced by $q_i$. Let $V_i = \big(V \cup \mathsf{vars}(F)\big) \cap \mathsf{vars}(q_i)$, and $Q_i = (q_i, V_i)$. By Sublemma 3.1, each $Q_i$ has an acyclic attack graph and by the induction hypothesis, we can assume a certain first-order rewriting $\varphi_i$ for $Q_i$ such that $\mathsf{qr}(\varphi_i) \leq \mathsf{diameter}(\tau_i)$ (for $1 \leq i \leq k$). Let $\Delta = \max_{1 \leq i \leq k} \mathsf{diameter}(\tau_i)$. Since $\mathsf{qr}(\bigwedge_{i=1}^{k} \varphi_i) = \max_{1 \leq i \leq k} \mathsf{qr}(q_i)$, it follows $\mathsf{qr}(\bigwedge_{i=1}^{k} \varphi_i) \leq \Delta$.

The crux is now that $\{q_1, \dots, q_k\}$ is an independent partition of $Q'$. Indeed, for $1 \leq i < j \leq k$, if $x \in \mathsf{vars}(q_i) \cap \mathsf{vars}(q_j)$, then by the definition of join tree, $x \in \mathsf{vars}(F)$, hence $x$ is a free variable in $Q'$. From Theorem 3.1, it follows that a certain first-order rewriting $\varphi_0$ for $Q$ can be obtained from the formula $\varphi$ in function `NaiveFo` by replacing the recursive call `NaiveFo`$(q', V')$ with $\bigwedge_{i=1}^{k} \varphi_i$. Since our construction guarantees $\mathsf{qr}(\varphi_0) \leq \ell + \mathsf{qr}(\bigwedge_{i=1}^{k} \varphi_i)$, we obtain $\mathsf{qr}(\varphi_0) \leq \ell + \Delta$. Finally, it can be easily seen that $\ell + \Delta \leq \mathsf{diameter}(\tau)$. This concludes the proof. $\square$

We illustrate in the following example the proof of Corollary 3.1. We show that the query of Example 3.9 has a certain first-order rewriting $\varphi$ with $\mathsf{qr}(\varphi) \leq 6$.

**Example 3.10** Consider query $q = \{R_0(\underline{x}), R_1(\underline{y}, x), R_2(\underline{x, y}), R_3(\underline{x}, a)\}$ of Example 3.9. Let $F = R_1(\underline{y}, x)$, an unattacked atom in the attack graph of $q$. Let $Q_1 = (\{R_0(\underline{x})\}, \{x\})$, $Q_2 = (\{R_2(\underline{x, y})\}, \{x, y\})$ and $Q_3 = (\{R_3(\underline{x}, a)\}, \{x\})$. Using Corollary 3.1, we compute the following certain first-order rewriting for $q$.

$$\begin{aligned}\varphi = \exists y \exists x \Big( \quad & R_1(\underline{y}, x) \wedge \forall x \Big( R_1(\underline{y}, x) \rightarrow \\ & \underbrace{R_0(\underline{x})}_{\text{rewriting of } Q_1} \wedge \underbrace{R_2(\underline{x, y})}_{\text{rewriting of } Q_2} \wedge \underbrace{R_3(\underline{x}, a) \wedge \forall z \big( R_3(\underline{x}, z) \rightarrow z = a \big)}_{\text{rewriting of } Q_3} \Big) \Big)\end{aligned}$$

We have $\mathsf{qr}(\varphi) = 4$. $\triangleleft$

Theorem 3.1 and Corollary 3.1 leads us to modify the function `NaiveFo` in order to get a certain first-order rewriting which decreases the quantifier (block) rank. This is function `SplitFo` on page 41.

**Example 3.11** Let $\rho_3$ and $\varphi_3$ be the queries of Example 3.8. It can be seen that $\mathsf{qbn}(\varphi_3) = \mathsf{qbr}(\varphi_3) = \mathsf{qr}(\varphi_3) = 6$. Using `SplitFo`, the query $\rho_3$ decreases the quantifier block rank and the quantifier rank: $\mathsf{qbr}(\rho_3) = \mathsf{qr}(\rho_3) = 2$. The number of quantifier blocks is not changed. Figure 3.1 shows the translations in SQL of $\varphi_3$ and $\rho_3$ respectively named `N3` and `R3`. The nesting depth of subqueries decreases from 5 in `N3` to 1 in `R3`. $\triangleleft$

## 3.2.3 Reduction of the Number of Quantifier Blocks

Function `NaiveFo` constructs a certain first-order rewriting by treating one unattacked atom at a time. However, if a query contains multiple unattacked atoms, then those atoms can be "rewritten" together. This is Theorem 3.2. Rewriting multiple unattacked atoms together generally results in less quantifier blocks, as expressed by Corollary 3.2. Examples 3.12 and 3.14 illustrate the idea.

**Example 3.12** We go on with the queries of Example 3.8. Remember the naive certain first-order rewriting $\varphi_3$ of $q_3$:

$$\varphi_3 = \begin{array}{l} \exists x_1 \Bigg[ R_1(\underline{x_1}, b) \wedge \forall y_1 \Bigg( R_1(\underline{x_1}, y_1) \rightarrow y_1 = b \wedge \\ \quad \exists x_2 \Bigg[ R_2(\underline{x_2}, b) \wedge \forall y_2 \Bigg( R_2(\underline{x_2}, y_2) \rightarrow y_2 = b \wedge \\ \quad\quad \exists x_3 \big[ R_3(\underline{x_3}, b) \wedge \forall y_3 \big( R_3(\underline{x_3}, y_3) \rightarrow y_3 = b \big) \big] \Bigg) \Bigg] \Bigg) \Bigg] \end{array}$$

It can be seen that atoms $R_1(\underline{x_1}, b)$, $R_2(\underline{x_2}, b)$ and $R_3(\underline{x_3}, b)$ are unattacked in the attack graph of $q_3$. An "improved" certain first-order rewriting $\beta_3$ for $q_3$ is given next.

$$\beta_3 = \begin{array}{l} \exists x_1 \exists x_2 \exists x_3 \Big( R_1(\underline{x_1}, b) \wedge R_2(\underline{x_2}, b) \wedge R_3(\underline{x_3}, b) \wedge \\ \quad \forall y_1 \forall y_2 \forall y_3 \big( R_1(\underline{x_1}, y_1) \wedge R_2(\underline{x_2}, y_2) \wedge R_3(\underline{x_3}, y_3) \rightarrow \\ \quad\quad y_1 = b \wedge y_2 = b \wedge y_3 = b \big) \Big) \end{array}$$

In $\beta_3$, the three unattacked atoms are "rewritten" together, resulting in only one implication with a conjunction of three atoms on both sides of the implication. This allows the quantifiers to be regrouped in blocks of the same type and results in a smaller number of quantifier blocks. Notice that the quantifier rank is still equal to 6 in $\beta_3$. $\triangleleft$

**Example 3.13** Consider query $q = \{R_0(\underline{y_1, y_2, y_3}), R_1(\underline{x_1}, y_1), R_2(\underline{x_2}, y_2), R_3(\underline{x_3}, y_3)\}$. To simplify the notation, let $F_i = R_i(\underline{x_i}, y_i)$ for $1 \leq i \leq 3$ and $G = R_0(\underline{y_1, y_2, y_3})$. The attack graph of $q$ is shown in Figure 3.4. Function `NaiveFo` or `SplitFo` will first choose,

```
Function SplitFo($q$,$V$) constructs certain first-order rewriting.
  Input: $Q = (q, V)$ is an acyclic SJFC query whose attack graph is acyclic and
         where $V$ is the set of free variables of $q$.
  Result: certain first-order rewriting $\rho$ for $Q$.
  begin
      if $q = \emptyset$ then
          $\rho \leftarrow$ true;
      else
          let $\{q_1, \ldots, q_m\}$ be a maximal partition of $q$ such that for $1 \leq i < j \leq m$,
          $\mathsf{vars}(q_i) \cap \mathsf{vars}(q_j) \subseteq V$;
          if $m \geq 2$ then
              foreach $i \leftarrow 1$ to $m$ do
                  $V_i' \leftarrow V \cap \mathsf{vars}(q_i)$
              $\rho \leftarrow \bigwedge_{i=1}^{m}$ SplitFo($q_i, V_i'$);
          else                                                          /* $m = 1$ */
              $E \leftarrow \mathsf{AttackGraph}((q, V))$;
              choose $F = R(\underline{x_1, \ldots, x_k}, y_1, \ldots, y_\ell)$ in $q$ such that $\forall G \in q : (G, F) \notin E$;
              $V' \leftarrow V$;
              $\mathsf{X} \leftarrow \emptyset$;
              foreach $i \leftarrow 1$ to $k$ do
                  if $x_i$ is a variable and $x_i \notin V'$ then
                      $V' \leftarrow V' \cup \{x_i\}$;
                      $\mathsf{X} \leftarrow \mathsf{X} \cup \{x_i\}$;
              $\mathsf{Y} \leftarrow \emptyset$;
              $\mathsf{NEW} \leftarrow \emptyset$;
              foreach $i \leftarrow 1$ to $\ell$ do
                  if $y_i$ is a constant or $y_i \in V'$ then
                      let $z_i$ be a new variable;
                      $\mathsf{NEW} \leftarrow \mathsf{NEW} \cup \{i\}$;
                  else                                     /* $y_i$ is a variable not in $V'$ */
                      let $z_i$ be the same variable as $y_i$;
                      $V' \leftarrow V' \cup \{y_i\}$;
                      $\mathsf{Y} \leftarrow \mathsf{Y} \cup \{y_i\}$;
              $q' \leftarrow q \setminus \{F\}$;
              $V' \leftarrow V' \cap \mathsf{vars}(q')$;
              $\rho \leftarrow \left[\begin{array}{l} \exists\mathsf{X}\,\exists\mathsf{Y} R(\underline{x_1, \ldots, x_k}, y_1, \ldots, y_\ell) \wedge \\ \forall z_1 \ldots \forall z_\ell \left[ R(\underline{x_1, \ldots, x_k}, z_1, \ldots, z_\ell) \rightarrow \left[\begin{array}{l} \bigwedge_{i \in \mathsf{NEW}} z_i = y_i \\ \wedge\ \mathtt{SplitFo}(q', V') \end{array}\right]\right] \end{array}\right]$;
      return $\rho$;
```

$$\begin{tikzcd}
R_1(\underline{x_1}, y_1) \arrow[dr, bend right] & R_2(\underline{x_2}, y_2) \arrow[d, bend left] & R_3(\underline{x_3}, y_3) \arrow[dl, bend left] \\
& R_0(\underline{y_1, y_2, y_3}) &
\end{tikzcd}$$

Figure 3.4: The attack graph of $q$ in Example 3.14.

for example, $F_1$, then $F_2$, then $F_3$ and eventually $G$, resulting in the following certain first-order rewriting $\varphi$.

$$\begin{aligned}
\varphi = \exists x_1 \exists y_1 \Bigg[ R_1(\underline{x_1}, y_1) \wedge \forall y_1' \Bigg( & R_1(\underline{x_1}, y_1') \rightarrow \\
& \Big[ \exists x_2 \exists y_2 \Big( R_2(\underline{x_2}, y_2) \wedge \forall y_2' \Big( R_2(\underline{x_2}, y_2') \rightarrow \\
& \quad \big[ \exists x_3 \exists y_3 \big( R_3(\underline{x_3}, y_3) \wedge \forall y_3' (R_3(\underline{x_3}, y_3') \rightarrow R_0(\underline{y_1', y_2', y_3'}))\big)\big] \Big) \Big) \Big] \Bigg) \Bigg]
\end{aligned}$$

Since $F_1, F_2$ and $F_3$ are unattacked in the attack graph of $q$, they can be "rewritten" together and an "improved" certain first-order rewriting can be computed, as follows.

$$\begin{aligned}
\beta = \exists x_1 \exists x_2 \exists x_3 \exists y_1 \exists y_2 \exists y_3 \Big( & R_1(\underline{x_1}, y_1) \wedge R_2(\underline{x_2}, y_2) \wedge R_3(\underline{x_3}, y_3) \wedge \\
& \forall y_1' \forall y_2' \forall y_3' \big( R_1(\underline{x_1}, y_1') \wedge R_2(\underline{x_2}, y_2') \wedge R_3(\underline{x_3}, y_3') \big) \\
& \qquad \rightarrow R_0(\underline{y_1', y_2', y_3'}) \Big)
\end{aligned}$$

It is easy to see that $\beta$ has a number of quantifier blocks that is less than the one of $\varphi$: $\mathsf{qbn}(\beta) = 2$ while $\mathsf{qbn}(\varphi) = 6$. Notice that $\varphi$ and $\beta$ both contain 9 quantifiers. $\lhd$

**Theorem 3.2** *Let $Q = (q, V)$ be an acyclic* **SJFC** *query. Let $S \subseteq q$ be a set of unattacked atoms in $Q$'s attack graph. Let $X = \Big(\bigcup_{F \in S} \mathsf{keyVars}(F)\Big) \setminus V$. If $\varphi$ is a certain first-order rewriting for $(q, V \cup X)$, then $\exists X \varphi$ is a certain first-order rewriting for $Q$.*

Proof. The proof makes use of the following sublemma.

**Sublemma 3.2** *[Wijsen 2010, Corollary 1] Let $Q = (q, V)$ be an acyclic* **SJFC** *query whose attack graph is acyclic. Let $F \in Q$ be an atom which is unattacked in the attack graph of $Q$. Assume without loss of generality that $\langle v_1, \ldots, v_m \rangle$ is the ordered sequence of variables of $V$ and that $\langle x_1, \ldots x_k \rangle$ is the ordered sequence of variables of $\mathsf{keyVars}(F) \setminus V$. Let $Q' = (q, V \cup \mathsf{keyVars}(F))$. Then, for every database $\mathsf{db}$, for every $\vec{a} \in \mathbf{dom}^m$, there exists $\vec{b} \in \mathbf{dom}^k$ such that $\vec{a} \in Q_{\mathsf{sure}}(\mathsf{db})$ implies $\langle \vec{a}, \vec{b} \rangle \in Q'_{\mathsf{sure}}(\mathsf{db})$.*

We show Theorem 3.2. Assume without loss of generality that $\langle v_1, \ldots, v_m, x_1, \ldots, x_\ell \rangle$ is the ordered sequence of variables of $V \cup X$, where $v_i \in V$ and $x_j \in X$ for $1 \leq i \leq m$, $1 \leq j \leq \ell$. Let $\vec{v} = \langle v_1, \ldots, v_m \rangle$ and $\vec{x} = \langle x_1, \ldots, x_\ell \rangle$. Let $Q' = (q, V \cup X)$. Assume $\varphi$ is a certain first-order rewriting for $(q, V \cup X)$. Then, for every database $\mathsf{db}$, for all $\vec{a} \in \mathbf{dom}^m$, $\vec{b} \in \mathbf{dom}^\ell$,

$$\langle \vec{a}, \vec{b} \rangle \in Q'_{\mathsf{sure}}(\mathsf{db}) \iff \mathsf{db} \models \varphi(\vec{a}, \vec{b}). \tag{3.1}$$

We need to show that for every database $\mathsf{db}$, for every $\vec{a} \in \mathbf{dom}^m$,

$$\vec{a} \in Q_{\mathsf{sure}}(\mathsf{db}) \iff \mathsf{db} \models \exists \vec{x} \varphi(\vec{a}, \vec{x}).$$

$\boxed{\Leftarrow}$ Easy. $\boxed{\Rightarrow}$ Assume $S = \{F_1, \ldots, F_u\}$. Let $X_1 = \mathsf{keyVars}(F_1) \setminus V$, and assume w.l.o.g. that $\vec{x}_1 = \langle x_1, \ldots, x_k \rangle$ with $k \leq \ell$ is the ordered sequence of variables in $X_1$. Let $Q^1 = (q, V \cup X_1)$. By Sublemma 3.2, for every database $\mathsf{db}$, for every $\vec{a} \in \mathbf{dom}^m$, there exists $\vec{b} \in \mathbf{dom}^k$ such that:

$$\vec{a} \in Q_{\mathsf{sure}}(\mathsf{db}) \implies \langle \vec{a}, \vec{b} \rangle \in Q^1{}_{\mathsf{sure}}(\mathsf{db}).$$

By Sublemma 3.1, none of the atoms of $S$ is attacked in the attack graph of $Q^1$. Then, by repeated application of the same arguments, for every database $\mathsf{db}$, for every $\vec{a} \in \mathbf{dom}^m$, there exists $\vec{b} \in \mathbf{dom}^\ell$ such that:

$$\vec{a} \in Q_{\mathsf{sure}}(\mathsf{db}) \implies \langle \vec{a}, \vec{b} \rangle \in Q'_{\mathsf{sure}}(\mathsf{db}). \tag{3.2}$$

From (3.1) and (3.2), for every database $\mathsf{db}$, for every $\vec{a} \in \mathbf{dom}^m$, there exists $\vec{b} \in \mathbf{dom}^\ell$ such that $\vec{a} \in Q_{\mathsf{sure}}(\mathsf{db})$ implies $\mathsf{db} \models \varphi(\vec{a}, \vec{b})$. Consequently, for every database $\mathsf{db}$, for every $\vec{a} \in \mathbf{dom}^m$, $\vec{a} \in Q_{\mathsf{sure}}(\mathsf{db})$ implies $\mathsf{db} \models \exists \vec{x} \varphi(\vec{a}, \vec{x})$. $\square$

**Corollary 3.2** *Let $Q = (q, V)$ be an acyclic* **SJFC** *query whose attack graph is acyclic. Let $p$ be the number of atoms on the longest directed path in the attack graph of $Q$. There exists a certain first-order rewriting $\varphi$ for $Q$ such that $\mathsf{qbn}(\varphi) \leq 2p$.*

Proof. For every $F \in q$, we define the *stratum* of $F$ as the number of atoms on the longest directed path (in the attack graph of $Q$) that starts from some unattacked atom and ends at $F$. In particular, an atom has stratum 1 if and only if it is unattacked. The greatest stratum is $p$.

From Theorem 3.2, it follows that function `NaiveFo` can be modified so as to proceed stratum-by-stratum (rather than atom-by-atom), starting with all atoms at stratum 1. In particular, let $\{R_i(\underline{\vec{x_i}}, \vec{y_i})\}_{i=1}^k$ be all atoms at stratum 1. Then $Q$ has a certain first-order rewriting of the form:

$$\exists \vec{x} \exists \vec{y} \Big( \bigwedge_{i=1}^{k} R_i(\underline{\vec{x_i}}, \vec{y_i}) \land \forall \vec{z} \big( \bigwedge_{i=1}^{k} R_i(\underline{\vec{x_i}}, \vec{z_i}) \rightarrow \phi \land \psi \big) \Big),$$

where the construction of $\vec{x}$, $\vec{y}$, and $\vec{z}$ is as in function `NaiveFo`, $\phi$ is a conjunction of equality predicates, and $\psi$ is a certain first-order rewriting of the query obtained from $Q$

by removing all atoms with stratum 1. If $\psi$ is obtained by recursive application of the same method (on a smaller query), we obtain a certain first-order rewriting for $Q$ that has a prenex normal form with $2p$ quantifier blocks. $\square$

The optimization is implemented in function `GroupingFo` and relies on Theorem 3.2. Function `GroupingFo` on page 45 improves function `NaiveFo` by decreasing the number of quantifier blocks of certain first-order rewritings.

**Example 3.14** Consider the queries $\varphi_3, \rho_3$ and $\beta_3$ of Examples 3.8 and 3.12. In particular, using Theorem 3.2, $\beta_3$ groups the quantifiers of the same type and decreases the number of quantifier blocks of $\varphi_3$ from 6 to 2. As already mentioned in Example 3.12, the quantifier ranks of $\varphi_3$ and $\beta_3$ are equal. The SQL queries `Q3,N3` and `B3` shown in Figure 3.1 encode respectively $q_3, \varphi_3$ and $\beta_3$. Query `N3` contains 5 `NOT EXISTS` subqueries while `B3` only contains 1 `NOT EXISTS` subquery. The following table summarizes those metrics.

| | (Naive) $\varphi_3$ | (Split) $\rho_3$ | (Grouping) $\beta_3$ |
|---|---|---|---|
| $\mathsf{qbn}(\cdot)$ | 6 | 6 | **2** |
| $\mathsf{qbr}(\cdot)$ | 6 | **2** | **2** |
| $\mathsf{qr}(\cdot)$ | 6 | **2** | 6 |
| depth of nested `NOT EXISTS` subqueries | 5 | **1** | **1** |
| number of `NOT EXISTS` subqueries | 5 | 3 | **1** |

$\lhd$

Corollaries 3.1 and 3.2 provide an upper bound on the quantifier block rank and on the number of quantifier blocks that are needed in certain first-order rewritings. We see that function `NaiveFo` can be easily modified so as to diminish either of those measures. In particular, `SplitFo` aims to reduce the quantifier (block) rank while `GroupingFo` aims to reduce the number of quantifier blocks.

One can see that the certain first-order rewritings computed using function `SplitFo` or function `GroupingFo` never increase the number of quantifier blocks nor the quantifier (block) rank in comparison with the certain first-order rewritings computed using `NaiveFo`. Indeed, no new quantifier is added when computing $\rho$ or $\beta$ in the last line of these functions. Moreover, the depth of recursive calls in `NaiveFo` is exactly the number of atoms in the query while the depth of recursive calls in `SplitFo` and `GroupingFo` can be smaller.

The three functions have a similar time complexity. Note that, in contrast with the complexity of $\mathsf{CERTAINTY}(q)$ which is about data complexity, the complexity of the three functions is about query complexity. A rough time complexity analysis shows that one recursive call is solved in $\mathcal{O}(|Q|^2)$, due to the complexity of function $\mathsf{AttackGraph}$ which runs in quadratic time in the length of $Q$ [Wijsen 2012]. As the number of recursive calls is bound by the number of atoms in the query, the resulting time complexity to compute a certain first-order rewriting is thus in $\mathcal{O}(|Q|^3)$.

The following table gives the number of quantifier blocks, the quantifier block rank and the quantifier rank of the certain first-order rewritings computed using the three functions introduced in this chapter. The input query is $\llbracket m \rrbracket$ of Example 3.4.

**Function** GroupingFo($q$,$V$) constructs certain first-order rewriting.

```
Input: $Q = (q, V)$ is an acyclic SJFC query whose attack graph is acyclic and
       where $V$ is the set of free variables of $q$.
Result: certain first-order rewriting $\beta$ for $Q$.
begin
    if $q = \emptyset$ then
        $\beta \leftarrow$ true;
    else
        $E \leftarrow \mathsf{AttackGraph}((q, V))$;
        Let $F_1, F_2, \ldots, F_u$ in $q$ such that $\forall G \in q \forall i \in \{1, \ldots, u\} : (G, F_i) \notin E$;
        $V' \leftarrow V$;
        $\mathsf{X} \leftarrow \emptyset$;
        $\mathsf{Y} \leftarrow \emptyset$;
        foreach $j \leftarrow 1$ to $u$ do
            Let $F_j = R_j(\underline{x_{j,1}, \ldots, x_{j,k}}, y_{j,1}, \ldots, y_{j,\ell})$;
            $\mathsf{NEW}_j \leftarrow \emptyset$;
            foreach $i \leftarrow 1$ to $k$ do
                if $x_{j,i}$ is a variable and $x_{j,i} \notin V'$ then
                    $V' \leftarrow V' \cup \{x_{j,i}\}$;
                    $\mathsf{X} \leftarrow \mathsf{X} \cup \{x_{j,i}\}$;
            foreach $i \leftarrow 1$ to $\ell$ do
                if $y_{j,i}$ is a constant or $y_{j,i} \in V'$ then
                    let $z_{j,i}$ be a new variable;
                    $\mathsf{NEW}_j \leftarrow \mathsf{NEW}_j \cup \{i\}$;
                else                         /* $y_{j,i}$ is a variable not in $V'$ */
                    let $z_{j,i}$ be the same variable as $y_{j,i}$;
                    $V' \leftarrow V' \cup \{y_{j,i}\}$;
                    $\mathsf{Y} \leftarrow \mathsf{Y} \cup \{y_{j,i}\}$;
        $q' \leftarrow q \setminus \{F_1, F2, \ldots, F_u\}$;
        $V' \leftarrow V' \cap \mathsf{vars}(q')$;
        $\beta \leftarrow \left[\begin{array}{l} \exists \mathsf{X}\, \exists \mathsf{Y} \bigwedge_{j=1}^{u} R_j(\underline{\vec{x_j}}, \vec{y_j}) \wedge \\ \forall \vec{z_1} \ldots \forall \vec{z_u} \left[\bigwedge_{j=1}^{u} R_j(\underline{\vec{x_j}}, \vec{z_j}) \rightarrow \left[\begin{array}{l} \bigwedge_{j=1}^{u} \bigwedge_{i \in \mathsf{NEW}_j} z_{j,i} = y_{j,i} \\ \wedge\ \mathtt{GroupingFo}(q', V') \end{array}\right]\right] \end{array}\right]$;
    return $\beta$;
```

| | `NaiveFo` | `SplitFo` | `GroupingFo` |
|---|---|---|---|
| $\mathsf{qbn}(\cdot)$ | $2m$ | $2m$ | $\mathbf{2}$ |
| $\mathsf{qbr}(\cdot)$ | $2m$ | $\mathbf{2}$ | $\mathbf{2}$ |
| $\mathsf{qr}(\cdot)$ | $2m$ | $\mathbf{2}$ | $2m$ |

However, we think it is generally not possible to minimize the three metrics simultaneously. For instance, it does not exist a certain first-order rewriting $\psi$ for $\llbracket m \rrbracket$ with $\mathsf{qbn}(\psi) = \mathsf{qbr}(\psi) = \mathsf{qr}(\psi) = 2$.

CHAPTER 4

# Certain Conjunctive Query Answering in SQL

For acyclic **SJFC** queries $q$, a certain first-order rewriting for $q$ exists if and only if the attack graph of $q$ is acyclic. Moreover, if a certain first-order rewriting exists, then it can be effectively computed. One of the main advantages of this approach is the ease to use the resulting first-order query in database applications. In this chapter, we first show how to compute certain first-order rewritings in SQL. Then, we investigate the performance of these SQL queries in practice.

In Section 4.1, we explain how one can obtain a certain SQL rewriting for a given query $q$. We do not directly translate from first-order logic to SQL: the certain rewritings are first computed in *tuple relational calculus* [Ullman 1988] (TRC), and then translated in SQL. Subsection 4.1.1 introduces the tuple relational calculus and three functions that compute certain TRC rewritings. Subsection 4.1.2 shows how to translate those rewritings from TRC to SQL.

We then investigate in Section 4.2 whether the certain SQL rewriting technique is an efficient practical technique to compute the certain answer of a query. We measure the execution times of several certain SQL rewritings and the impact of the following parameters:

- the syntactic optimizations presented in the previous chapter,
- the fraction of tuples involved in some primary key violation,
- the size of the database, and
- the cardinality of the query.

The experiments, the measurements and the observations are summarized in Section 4.2.

## 4.1 From First-Order to SQL

The first-order queries introduced in Sections 3.1 and 3.2 were expressed in *domain relational calculus* (DRC), a first-order language in which the variables range over the constants of the domain. It is usual in the database field to refer to queries that are in DRC with the terms *first-order queries*. Given an acyclic **SJFC** query $q$ whose attack graph is acyclic, the procedures described in these sections compute a certain first-order rewriting $\varphi$ in DRC. If we want to execute $\varphi$ using standard DBMS technology, it has to be translated in SQL.

A certain SQL rewriting can be obtained by a direct translation from first-order logic to SQL, but the technical treatment involved is difficult. In this section, we present three functions that return certain rewritings in *tuple relational calculus* (TRC) [Ullman 1988]. The motivation is that the translation from TRC to SQL is straightforward.

The whole process is summarized by Figure 4.1. Given an acyclic **SJFC** query $q$ in first-order logic, we compute a certain SQL rewriting. The theory, including the syntactic optimizations, is made on first-order queries (left part of the figure). To compute a certain SQL rewriting for $q$, we first compute a certain TRC rewriting for $q$ using one of the three new functions that produce a certain TRC rewriting and the resulting query is then translated in SQL (right part of the figure, in red).

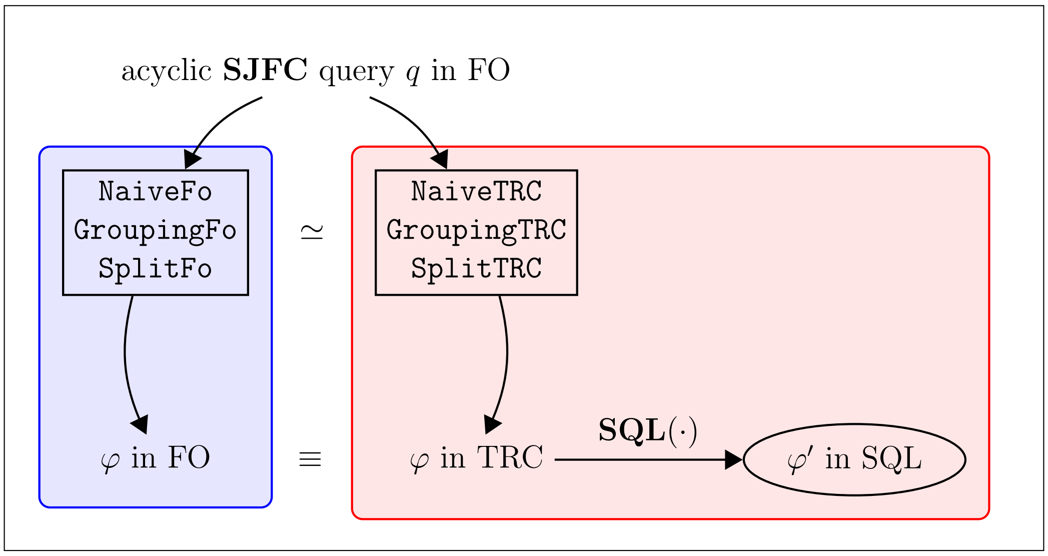

Figure 4.1: From a first-order acyclic **SJFC** query $q$ to a certain SQL rewriting $\varphi'$ for $q$.

### 4.1.1 Tuple Relational Calculus

The set of legal tuple relational calculus formulas we consider is defined in a way that is similar to domain relational calculus formulas, except for the basic atoms, which are of the following form [Maier 1983]:

1. For any relation name $R$ whose arity is $n$, and for any tuple variable $t$ with the same arity, $R(t)$ is an atom and is true if $t \in R$. The arity of $t$ is denoted by $\mathsf{arity}(t)$;

2. For any tuple variables $t_1, t_2$, and for any positions $i, j$ (with $1 \leq i \leq \mathsf{arity}(t_1)$ and $1 \leq j \leq \mathsf{arity}(t_2)$), $t_1 \cdot i \ = \ t_2 \cdot j$ is an atom;

3. For any tuple variable $t$, any position $i$ (with $1 \leq i \leq \mathsf{arity}(t)$) and if $c$ is a constant in the domain, then $c \ = \ t \cdot i$ and $t \cdot i \ \theta \ c$ are atoms.

The following abbreviations will be used.

- $\forall t \in R(\varphi)$ is a shorthand for $\forall t\big(R(t) \rightarrow \varphi\big)$, and
- $\exists t \in R(\varphi)$ is a shorthand for $\exists t\big(R(t) \wedge \varphi\big)$.

The translation from first-order logic to TRC is straightforward for conjunctive queries, as illustrated by the following example.

**Example 4.1** Let $q \ = \ \exists x_1 \exists x_2 \exists x_3 R_1(\underline{x_1, x_2}, a) \wedge R_2(\underline{x_2}, x_3)$. An equivalent formula in TRC is given by $q'$:

$$q' = \exists t_1 \exists t_2 \big(R_1(t_1) \wedge R_2(t_2) \wedge t_1 \cdot 2 = t_2 \cdot 1 \wedge t_1 \cdot 3 = a\big)$$

The variables range over the tuples of their respective relations, $t_1$ ranges over the tuples of $R_1$ and $t_2$ ranges over the tuples of $R_2$. $\triangleleft$

The functions `NaiveFo, GroupingFo` and `SplitFo` have been modified to output queries that are in TRC. The resulting functions are respectively `NaiveTRC`, `GroupingTRC` and `SplitTRC`. Importantly, the functions take as input an acyclic **SJFC** query $Q = (q, V)$ expressed in first-order logic. The output is in TRC.

The function `NaiveTRC` is shown on page 51. It initializes some data structures, and then makes a call to the subroutine `NaiveTRCSub`. The function `GroupingTRC` is not shown, because it is the same as `NaiveTRC` up to changing the call of `NaiveTRCSub` with a call to `GroupingTRCSub`. The latter subroutine is shown on page 53. Likewise, the function `SplitTRC` is not shown, because it can be obtained from `NaiveTRC` by replacing the call of `NaiveTRCSub` with `SplitTRCSub`. The subroutine `SplitTRCSub` is shown on page 54.

For every relation name $R$ (in upper case), we introduce three tuple variables $r_0, r_1$ and $r_2$ (in lower case) that range over relation $R$. The parameter $\mathsf{REFS}$ is an array whose indices are the variables of $\mathsf{vars}(q)$ and the value is of the form $r \cdot i$ where $r$ is a variable name and $i$ a position in $r$.

The initialization in `NaiveTRC`, `GroupingTRC` or `SplitTRC` essentially handles the free variables of the input query $Q = (q, V)$. For every free variable $x$ of $Q$ (i.e. for every variable $x \in V$), the initialization picks an atom of $q$ that contains $x$. If this atom is $R(x_1, x_2, \ldots, x_n)$ with $x_i = x$, then $\mathsf{REFS}[x]$ is initialized to $r_0 \cdot i$. Entries in $\mathsf{REFS}$ will never be changed after their initialization.

**Example 4.2** Let $R, S$ be relation names with signature $[2, 1]$. Let $Q = (q, V)$ with $V = \{x\}$ and $q = \{R(\underline{a}, y), S(\underline{y}, x)\}$. Consider for example a run of `SplitTRC`$(q, V)$.

The only free variable $x$ occurs at the second position in $S$. Function `SplitTRC` initializes $\mathsf{REFS}$ such that $\mathsf{REFS}[x] = s_0 \cdot 2$. A certain TRC rewriting for $Q$ is of the form $\{s_0 \cdot 2 \mid s_0 \in S \wedge \varphi\}$ where $\varphi$ is the output of `SplitTRCsub`$(q, V, \mathsf{REFS})$. $\triangleleft$

We now explain a run of function `SplitTRCsub`.

**Example 4.3** We go on with Example 4.2. A call to $\texttt{SplitTRCsub}(q, V, \mathsf{REFS})$ chooses the unattacked atom $R(\underline{a}, y)$ and eventually returns

$$\begin{aligned}\exists r_1 \in R\Big(\, & r_1 \cdot 1 = a \wedge \\ & \forall r_2 \in R\big(\, r_2 \cdot 1 = r_1 \cdot 1 \rightarrow \\ & \qquad \mathbf{true} \wedge \texttt{SplitTRCsub}(q', \{x, y\}, \mathsf{REFS})\big)\Big),\end{aligned}$$

with $q' = \{S(\underline{y}, x)\}$, $\mathsf{REFS}[x] = s_0 \cdot 2$ (unchanged), and $\mathsf{REFS}[y] = r_2 \cdot 2$. The call $\texttt{SplitTRCsub}(q', \{x, y\}, \mathsf{REFS})$ chooses the only remaining atom $S(\underline{y}, x)$ and eventually returns

$$\begin{aligned}\exists s_1 \in S\Big(\, & s_1 \cdot 1 = r_2 \cdot 2 \wedge \\ & \forall s_2 \in S\big(\, s_2 \cdot 1 = s_1 \cdot 1 \rightarrow \\ & \qquad s_2 \cdot 2 = s_0 \cdot 2 \wedge \mathbf{true}\big)\Big).\end{aligned}$$

Putting everything together, we get a certain TRC rewriting for $Q$:

$$\begin{aligned}\{s_0 \cdot 2 \mid s_0 \in S \wedge \exists r_1 \in R\Big(\, & r_1 \cdot 1 = a \wedge \\ & \forall r_2 \in R\Big(r_2 \cdot 1 = r_1 \cdot 1 \rightarrow \\ & \qquad \exists s_1 \in S\big(\, s_1 \cdot 1 = r_2 \cdot 2 \wedge \\ & \qquad\qquad \forall s_2 \in S(\, s_2 \cdot 1 = s_1 \cdot 1 \rightarrow \\ & \qquad\qquad\qquad s_2 \cdot 2 = s_0 \cdot 2)\big)\Big)\Big)\}\end{aligned}$$

$\triangleleft$

```
Function NaiveTRC($q$,$V$)
Input: $Q = (q, V)$ is an acyclic SJFC query whose attack graph is acyclic and
       where $V$ is the set of free variables of $q$.
Result: A certain TRC rewriting $\varphi$ for $Q$
begin
    foreach variable $x$ in $q$ do
        $\mathsf{REFS}[x] \leftarrow \bot$;
    $\varphi \leftarrow$ true;
    $\mathsf{SELECT} \leftarrow \langle\rangle$;
    $\mathsf{USED} \leftarrow \emptyset$;
    foreach $x \in V$ do
        Let $F = R(\vec{x})$ be an atom in which $x$ occurs at position $i$;
        $\mathsf{REFS}[x] \leftarrow r_0 \cdot i$;
        append $r_0.i$ to $\mathsf{SELECT}$;
        if $R \notin \mathsf{USED}$ then
            $\varphi \leftarrow \varphi \wedge (r_0 \in R)$;        /* $r_0$ ranges over $R$-facts */
            $\mathsf{USED} \leftarrow \mathsf{USED} \cup \{R\}$;
    $\varphi \leftarrow \{\mathsf{SELECT} \mid \varphi \wedge \mathtt{NaiveTRCsub}(q, V, \mathsf{REFS})\}$;
    return $\varphi$;
```

### 4.1.2 Encoding from TRC to SQL

The functions introduced in the previous subsection output certain TRC rewritings. We briefly explain the main ideas and rules to translate the resulting queries in SQL.

Let **toSQL** denote the translation function. This function takes as input a query expressed in tuple relational calculus and returns a query which is semantically equivalent but expressed in SQL.

The translation into SQL is made recursively on the elements of the syntax tree of the input query. The main rules used for this function are described in Table 4.1. The function **ATTR**$(t, i)$ used for $t \cdot i$ is a function that returns the name of the $i$-th attribute of the relation of $t$. In practice, function **toSQL** is enhanced to handle some simple logical simplifications (like **true** $\wedge\, \varphi$ which becomes $\varphi$) and to support nonBoolean queries. This function constructs SQL queries following SQL99 standard. For example, we use `NOT EXISTS` instead of *forall*. Although SQL99 is a standard, it is not always strictly implemented by commercial DBMS's. For instance, MySQL does not support the `INTERSECT` operator. In our experiments, we replaced this operator by an appropriate join.

**Example 4.4** Let $\varphi$ be the certain first-order rewriting obtained in Example 4.3. An SQL translation using the (enhanced version of) function **toSQL** returns the following query.

```
SELECT s0.2 FROM S AS s0
WHERE EXISTS (SELECT *
              FROM R AS r1
```

**Function** NaiveTRCsub($q$,$V$,REFS)

```
Input: $Q = (q, V)$ is an acyclic SJFC query whose attack graph is acyclic and
       where $V$ is the set of free variables of $q$. REFS is an array whose indices are
       the variables of $V$ such that for each $x \in V$, REFS$[x]$ is of the form $r_j \cdot i$
       where $r_j$ is a tuple variable and $i$ a position.
Result: formula $\varphi$ in TRC
begin
    if $q = \emptyset$ then
        $\varphi \leftarrow$ true
    else
        $E \leftarrow$ AttackGraph$((q, V))$;
        choose $F = R(\underline{x_1, \ldots, x_k}, y_{k+1}, \ldots, y_\ell)$ in $q$ such that $\forall G \in q : (G, F) \notin E$;
        CONDX $\leftarrow$ true; CONDY $\leftarrow$ true;
        FREE $\leftarrow V$;
        foreach $i \leftarrow 1$ to $k$ do
            if $x_i$ is a constant then
                CONDX $\leftarrow$ CONDX $\wedge \big(r_1 \cdot i = x_i\big)$;
            else                                         /* $x_i$ is a variable */
                if $x_i \in$ FREE then
                    CONDX $\leftarrow$ CONDX $\wedge \big(r_1 \cdot i =$ REFS$[x_i]\big)$;
                else
                    FREE $\leftarrow$ FREE $\cup \{x_i\}$; REFS$[x_i] \leftarrow r_1 \cdot i$;
        foreach $i \leftarrow k+1$ to $\ell$ do
            if $y_i$ is a constant then
                CONDY $\leftarrow$ CONDY $\wedge \big(r_2 \cdot i = y_i\big)$;
            else                                         /* $y_i$ is a variable */
                if $y_i \in$ FREE then
                    CONDY $\leftarrow$ CONDY $\wedge \big(r_2 \cdot i =$ REFS$[y_i]\big)$;
                else
                    FREE $\leftarrow$ FREE $\cup \{y_i\}$; REFS$[y_i] \leftarrow r_2 \cdot i$;
        $q' \leftarrow q \setminus \{F\}$;
        $\varphi \leftarrow \left[\begin{array}{l} \exists r_1 \in R\Big[\text{CONDX} \wedge \forall r_2 \in R\ \Big(\bigwedge_{i=1}^{k}(r_2 \cdot i = r_1 \cdot i) \rightarrow \\ \qquad\qquad \text{CONDY} \wedge \texttt{NaiveTRCsub}(q', \text{FREE}, \text{REFS})\Big)\Big] \end{array}\right]$;
    return $\varphi$
```

```
Function GroupingTRCsub($q$,$V$,REFS)
  Input: $Q = (q, V)$ is an acyclic SJFC query whose attack graph is acyclic and
         where $V$ is the set of free variables of $q$. REFS is an array whose indices are
         the variables of $V$ such that for each $x \in V$, REFS$[x]$ is of the form $r_j \cdot i$
         where $r_j$ is a tuple variable and $i$ a position.
  Result: formula $\beta$ in TRC
  begin
      if $q = \emptyset$ then
          $\beta \leftarrow$ true
      else
          $E \leftarrow$ AttackGraph$((q, V))$;
          Let $F_1, F_2, \dots, F_u$ in $q$ such that $\forall G \in q \forall i \in \{1, \dots, u\} : (G, F_i) \notin E$;
          $\beta \leftarrow$ true; CONDX $\leftarrow$ true; CONDY $\leftarrow$ true;
          FREE $\leftarrow V$; KEY $\leftarrow []$;
          foreach $j \leftarrow 1$ to $u$ do
              Let $F_j = R_j(\underline{x_{j,1}, \dots, x_{j,k}}, y_{j,k+1}, \dots, y_{j,\ell})$;
              KEY$[R_j] \leftarrow k$;
              foreach $i \leftarrow 1$ to $k$ do
                  if $x_{j,i}$ is a constant then
                      CONDX $\leftarrow$ CONDX $\wedge \big(r_{j,1} \cdot i = x_{j,i}\big)$;
                  else                                        /* $x_{j,i}$ is a variable */
                      if $x_{j,i} \in$ FREE then
                          CONDX $\leftarrow$ CONDX $\wedge \big(r_{j,1} \cdot i =$ REFS$[x_{j,i}]\big)$;
                      else
                          FREE $\leftarrow$ FREE $\cup \{x_{j,i}\}$; REFS$[x_{j,i}] \leftarrow r_{j,1} \cdot i$;
              foreach $i \leftarrow k + 1$ to $\ell$ do
                  if $y_{j,i}$ is a constant then
                      CONDY $\leftarrow$ CONDY $\wedge \big(r_{j,2} \cdot i = y_{j,i}\big)$;
                  else                                        /* $y_{j,i}$ is a variable */
                      if $y_{j,i} \in$ FREE then
                          CONDY $\leftarrow$ CONDY $\wedge \big(r_{j,2} \cdot i =$ REFS$[y_{j,i}]\big)$;
                      else
                          FREE $\leftarrow$ FREE $\cup \{y_{j,i}\}$; REFS$[y_{j,i}] \leftarrow r_{j,2} \cdot i$;
          $q' \leftarrow q \setminus \{F_1, \dots, F_u\}$;
          $\beta \leftarrow \left[\begin{array}{l} \exists r_{1,1} \in R_1 \dots \exists r_{u,1} \in R_u,\ \text{CONDX} \wedge \\ \quad \forall r_{1,2} \in R_1 \dots \forall r_{u,2} \in R_u \Big[ \bigwedge_{j=1}^{u} \bigwedge_{i=1}^{\text{KEY}[R_j]} (r_{j,2} \cdot i = r_{j,1} \cdot i) \\ \qquad \rightarrow \big[\text{CONDY} \wedge \texttt{GroupingTRCsub}(q', \text{FREE}, \text{REFS})\big]\Big] \end{array}\right]$;
      return $\beta$
```

**Function** SplitTRCsub($q$,$V$,REFS)

```
Input: $Q = (q, V)$ is an acyclic SJFC query whose attack graph is acyclic and
       where $V$ is the set of free variables of $q$. REFS is an array whose indices are
       the variables of $V$ such that for each $x \in V$, REFS$[x]$ is of the form $r_j \cdot i$
       where $r_j$ is a tuple variable and $i$ a position.
Result: formula $\rho$ in TRC
begin
    if $q = \emptyset$ then
        $\rho \leftarrow$ true
    else
        let $\{q_1, \ldots, q_m\}$ be a maximal partition of $q$ such that for $1 \leq i < j \leq m$,
        vars$(q_i) \cap$ vars$(q_j) \subseteq V$;
        if $m \geq 2$ then
            $\rho \leftarrow \bigwedge_{i=1}^{m}$ SplitTRCsub$(q_i, V, \text{REFS})$;
        else                                                          /* $m = 1$ */
            choose $F = R(\underline{x_1, \ldots, x_k}, y_{k+1}, \ldots, y_\ell)$ in $q$ such that
            $\forall G \in q : (G, F) \notin E$;
            CONDX $\leftarrow$ true; CONDY $\leftarrow$ true;
            FREE $\leftarrow V$;
            foreach $i \leftarrow 1$ to $k$ do
                if $x_i$ is a constant then
                    CONDX $\leftarrow$ CONDX $\wedge \big(r_1 \cdot i = x_i\big)$;
                else                                          /* $x_i$ is a variable */
                    if $x_i \in$ FREE then
                        CONDX $\leftarrow$ CONDX $\wedge \big(r_1 \cdot i = \text{REFS}[x_i]\big)$;
                    else
                        FREE $\leftarrow$ FREE $\cup \{x_i\}$; REFS$[x_i] \leftarrow r_1 \cdot i$;
            foreach $i \leftarrow k+1$ to $\ell$ do
                if $y_i$ is a constant then
                    CONDY $\leftarrow$ CONDY $\wedge \big(r_2 \cdot i = y_i\big)$;
                else                                          /* $y_i$ is a variable */
                    if $y_i \in$ FREE then
                        CONDY $\leftarrow$ CONDY $\wedge \big(r_2 \cdot i = \text{REFS}[y_i]\big)$;
                    else
                        FREE $\leftarrow$ FREE $\cup \{y_i\}$; REFS$[y_i] \leftarrow r_2 \cdot i$;
            $q' \leftarrow q \setminus \{F\}$;
            $\rho \leftarrow \left[\begin{array}{c} \exists r_1 \in R\Big[\text{CONDX} \wedge \forall r_2 \in R\Big(\bigwedge_{i=1}^{k}(r_2 \cdot i = r_1 \cdot i) \rightarrow \\ \text{CONDY} \wedge \text{SplitTRCsub}(q', \text{FREE}, \text{REFS})\Big)\Big] \end{array}\right]$;
    return $\rho$
```

| Input | Rule to apply |
|---|---|
| **true** | TRUE |
| **false** | FALSE |
| $c$ | "$c$" ($c$ is a constant) |
| $t$ | t ($t$ is a tuple variable) |
| $t.i$ | t.$\mathbf{ATTR}(t, i)$ $(1 \leq i \leq \mathsf{arity}(t))$ |
| $\exists t \in R\ T_1$ | EXISTS (SELECT * FROM R as t WHERE ($\mathbf{toSQL}(T_1)$)) |
| $\forall t \in R\ T_1$ | $\mathbf{toSQL}(\neg\exists t \in R\ \neg T_1)$ |
| $T_1 \rightarrow T_2$ | $\mathbf{toSQL}(T_1 \vee \neg T_2)$ |
| $T_1 \wedge \cdots \wedge T_n$ | ($\mathbf{toSQL}(T_1)$ AND ... AND $\mathbf{toSQL}(T_n)$) |
| $T_1 \vee \cdots \vee T_n$ | ($\mathbf{toSQL}(T_1)$ OR ... OR $\mathbf{toSQL}(T_n)$) |
| $\neg T_1$ | (NOT $\mathbf{toSQL}(T_1)$) |
| $T_1 = T_2$ | ($\mathbf{toSQL}(T_1) = \mathbf{toSQL}(T_2)$) |
| $T_1 \neq T_2$ | ($\mathbf{toSQL}(T_1)$ <> $\mathbf{toSQL}(T_2)$) |
| $T_1$ | SELECT 'true' WHERE ($\mathbf{toSQL}(T_1)$) |

Table 4.1: Main rules to translate a query from TRC to SQL.

```
WHERE r1.1 = 'a'
AND NOT EXISTS (SELECT *
                FROM R AS r2
                WHERE r2.1 = r1.1
                AND NOT EXISTS (SELECT *
                                FROM S as s1
                                WHERE s1.1 = r2.2
                                AND NOT EXISTS (SELECT *
                                                FROM S AS s2
                                                WHERE s2.1 = s1.1
                                                AND s2.2 <> s0.2))))
```

The above SQL query can be shortened by replacing `WHERE EXISTS` with a join:

```
SELECT s0.2 FROM S AS s0, R AS r1
WHERE r1.1 = 'a'
AND NOT EXISTS (SELECT *
                FROM R AS r2
                WHERE r2.1 = r1.1
                AND NOT EXISTS (SELECT *
                                FROM S as s1
                                WHERE s1.1 = r2.2
                                AND NOT EXISTS (SELECT *
                                                FROM S AS s2
                                                WHERE s2.1 = s1.1
                                                AND s2.2 <> s0.2)))
```

◁

## 4.2 Experiments

We showed how to compute certain first-order rewritings for acyclic **SJFC** queries whose attack graph is acyclic. We gave a function that produces a "naive" certain TRC rewriting and we showed how to get an SQL translation of this rewriting. We also provided two syntactic optimizations that aim to reduce the number of quantifier blocks and the quantifier (block) rank of certain first-order rewritings.

In this section, we investigate whether the certain query rewriting technique is an efficient way to compute the certain answer to a given query. In particular, we conducted experiments to answer the following questions:

- Does the certain rewriting technique result in *acceptable* evaluation times on practical databases? Is this technique efficient and applicable in a practical environment?

- Does the reduction of the quantifier block rank or the number of quantifier blocks result in better performance? Do the syntactic optimizations have an impact? Is this impact systematic?

- What affects the evaluation time of the different rewritings? How does the evaluation time behave in function of the proportion of uncertainties, of the size of the database, or of the size of the conjunctive query?

No established benchmark for certain query answering exists in the literature. The following subsections describe the two main experiments we conducted in order to answer the aforementioned questions. The first experiment involves a large subset of a practical database: (a snapshot of) the Wikipedia's database. In this first experiment, we study the feasability of the certain rewriting technique for both Boolean and nonBoolean queries. We measure the evaluation times for several variants of this database in which the number of tuples involved in a primary key violation differs.

The second experiment concerns several synthetic databases. This situation is less relevant to study the practicability of the certain rewriting approach but allows to specifically study the impact of the two syntactic optimizations considered so far. We also study the impact on the evaluation time of the databases size and of the size of the queries.

All the experiments were conducted on the same configuration: the hardware is an Intel Core i5 2540M running at 2.50Ghz with 6 gygabites of DDR3 memory. The hard disk is a Solid State Disk Crucial M4. The DBMS is MySQL 5.5.24 running on a Kubuntu 12.04 (kernel 3.2.0-32-generic, x64).

The following notational convention will be used in the following subsections: if $Q$ is an acyclic **SJFC** query whose attack graph is acyclic, then $\varphi_Q$ denotes the certain first-order rewriting for $Q$ obtained using `NaiveFo`, $\beta_Q$ denotes the certain first-order rewriting for $Q$ obtained using `GroupingFo` and $\rho_Q$ denotes the certain first-order rewriting for $Q$ obtained using `SplitFo`.

Finally, given a database `db` and a query $Q$, we denote by $T_{Q,\mathtt{db}}$ the average of 100 execution times of the (SQL translation of) $Q$ on `db`. It is important to note that the query cache is cleared after each execution of the query. Given a certain first-order rewriting $q'$ for $q$, we first compute a certain TRC rewriting $q''$ for $q$ using one of the

| Schema | Arity | Number of tuples | Size |
|---|---|---|---|
| $\mathsf{PAGE}[\underline{\mathsf{id}}, \mathsf{namespace}, \mathsf{title}, \ldots]$ | 12 | 4,862,082 | 417 MB |
| $\mathsf{CATEGORY}[\underline{\mathsf{id}}, \mathsf{title}, \ldots]$ | 6 | 296,002 | 13 MB |
| $\mathsf{CATEGORYLINKS}[\underline{\mathsf{from}, \mathsf{to}}, \ldots]$ | 7 | 14,101,121 | 1.8 GB |
| $\mathsf{INTERWIKI}[\underline{\mathsf{prefix}}, \mathsf{url}, \ldots]$ | 6 | 662 | 40 KB |
| $\mathsf{EXTERNALLINKS}[\underline{\mathsf{from}, \mathsf{to}}, \ldots]$ | 3 | 6,933,703 | 1.3 GB |

Table 4.2: Database schema of the first experiment.

functions `NaiveTRC`, `GroupingTRC` or `SplitTRC` and then translate the resulting certain TRC rewriting in SQL for execution.

### 4.2.1 Performances on a Practical Database

This first experiment aims to verify that computing the certain answer using the approach of certain query rewriting is a feasible approach, meaning that the time required to compute the certain answer is acceptable and scales well with the database size.

We are interested in measuring the execution times on large databases of practical interest. For this purpose, we choose a large subset of Wikipedia's database. We choose two conjunctive queries—a Boolean and a nonBoolean one—and measure their evaluation times on this database. Since SQL rewritings contain nested subqueries, we expect those evaluation times to be significantly higher for the three certain SQL rewritings compared to the conjunctive query.

For the second part of this experiment, we gradually add to the database an increasing number of tuples that violate one of the primary keys.

#### Databases and Queries

We use a subset of a snapshot of the relational database containing Wikipedia's meta-data which is publicly available at `http://dumps.wikimedia.org/frwiki/20120117/`.

The subset contains approximately 25 million tuples; its size is 3.5 gigabytes on disk. This subset is composed of 5 relations with 34 attributes. The database schema and the database size are shown in Table 4.2. Attributes that are not shown are not relevant for our queries. The full database layout is documented on Mediawiki's website[1].

We created six versions of this database in which we added tuples that are involved in some primary key violations. We will refer to those databases using the notation $\mathcal{W}_p$ where $p$ is the percentage of tuples added. To allow primary key violations, all primary key constraints were dropped and replaced by nonunique indexes. We introduce conflicting tuples as follows:

1. We randomly take two distinct tuples in each relation $R$. Let $R(\underline{\vec{a}}, \vec{b})$ and $R(\underline{\vec{c}}, \vec{d})$ (with $R(\underline{\vec{a}}, \vec{d})$ not in $R$) be those atoms;
2. We add a tuple $R(\underline{\vec{a}}, \vec{d})$ in $R$.

[1]Mediawiki's website is available at `http://www.mediawiki.org/wiki/Manual:Database_layout`.

The databases $\mathcal{W}_{0.001\%}, \mathcal{W}_{0.01\%}, \mathcal{W}_{0.1\%}, \mathcal{W}_{1\%}$ and $\mathcal{W}_{2\%}$ contain respectively 0.001%, 0.01%, 0.1%, 1% and 2% more tuples than $\mathcal{W}_{0\%}$. As no tuples are removed in order to add primary key violations, notice that the size of $\mathcal{W}_{2\%}$ for example, is 1.02 times the size of $\mathcal{W}_{0\%}$. As every new tuple added in the database is involved in a primary key violation, the proportion of conflicting tuples in $\mathcal{W}_{2\%}$ is $\frac{2\times 2}{1.02}\%$ (approximatively 4%). Although only 4% of uncertainties may seem small in a database, it still represents about 1.000.000 tuples involved in a primary key violation in our case.

In our experiments, we chose two acyclic **SJFC** queries which are in accordance with the intended semantics of the database schema. We investigate the performance of the evaluation of (certain SQL rewritings for) a Boolean query ($Q_B$) and a nonBoolean query ($Q_{nB}$):

$$Q_B = \exists x \exists t \exists y \exists u \exists \ldots \left[\begin{array}{l} \mathsf{PAGE}(\underline{x}, \ldots) \wedge \mathsf{CATEGORYLINKS}(\underline{x, t}, \ldots) \wedge \\ \mathsf{CATEGORY}(\underline{y}, t, \ldots) \wedge \mathsf{EXTERNALLINKS}(\underline{x, u}, \ldots) \wedge \\ \mathsf{INTERWIKI}(\underline{fr}, u, \ldots) \end{array}\right]$$

$$Q_{nB}(u) = \exists x \exists t \exists y \exists \ldots \left[\begin{array}{l} \mathsf{PAGE}(\underline{x}, \ldots) \wedge \mathsf{CATEGORYLINKS}(\underline{x, t}, \ldots) \wedge \\ \mathsf{CATEGORY}(\underline{y}, t, \ldots) \wedge \mathsf{EXTERNALLINKS}(\underline{x, u}, \ldots) \wedge \\ \mathsf{INTERWIKI}(\underline{fr}, u, \ldots) \end{array}\right]$$

The query $Q_B$ asks: "Is there some page with a category link to some category and with an external link to the Wiki identified by fr?" and evaluates to true on each database. Query $Q_{nB}$ is a nonBoolean variant of query $Q_B$ and asks for the addresses of those Wikis identified by fr.

To help understanding, we point out that attribute $\mathsf{CATEGORYLINKS} \cdot \mathsf{to}$ refers to $\mathsf{CATEGORY} \cdot \mathsf{title}$, not to $\mathsf{CATEGORY} \cdot \mathsf{id}$. Each position that is not shown contains a new distinct variable which does not occur elsewhere.

The queries $\varphi_{Q_B}, \beta_{Q_B}$ and $\rho_{Q_B}$ (resp. $\varphi_{Q_{nB}}, \beta_{Q_{nB}}$ and $\rho_{Q_{nB}}$) refer to the certain first-order rewritings for $Q_B$ (resp. $Q_{nB}$), using `NaiveFo, GroupingFo` and `SplitFo`. For instance, SQL translations of queries $\rho_{Q_B}$ and $\beta_{Q_{nB}}$ are shown in Figure 4.2. The following table gives the number of subqueries and the maximal depth of nested subqueries for the SQL translations of the three certain first-order rewritings for $Q_B$ and $Q_{nB}$.

| | $\varphi_{Q_B}$ | $\beta_{Q_B}$ | $\rho_{Q_B}$ | $\varphi_{Q_{nB}}$ | $\beta_{Q_{nB}}$ | $\rho_{Q_{nB}}$ |
|---|---|---|---|---|---|---|
| number of (NOT) EXISTS subqueries | 8 | 4 | 6 | 8 | 4 | 6 |
| depth of nested (NOT) EXISTS subqueries | 8 | 4 | 5 | 8 | 4 | 5 |

## Observations and Measurements

Each of the eight queries is evaluated against the six different databases. The resulting evaluation times are summarized in Table 4.3. The execution times of the certain SQL rewritings on large databases seem acceptable in practice. A further analysis reveals the following:

1. Despite the size of the databases, the execution times are very small. The execution time ranges from 1.15 milliseconds to 6.16 milliseconds.

```
RHO_QB = SELECT 'true' FROM interwiki AS iw
     WHERE (iw.iw_prefix = 'fr'
     AND (NOT EXISTS
         (SELECT * FROM interwiki AS iw_bis
          WHERE ((NOT EXISTS
             (SELECT * FROM category AS c
              WHERE (NOT EXISTS
                 (SELECT * FROM category AS c_bis
                  WHERE ((NOT EXISTS
                     (SELECT * FROM externallinks AS el
                      WHERE (iw_bis.iw_url = el.el_to
                         AND EXISTS
                           (SELECT * FROM categorylinks AS cl
                            WHERE (el.el_from = cl.cl_from
                               OR c_bis.cat_title = cl.cl_to))
                         AND EXISTS
                           (SELECT * FROM page AS p
                            WHERE el.el_from = p.page_id))))
                     AND c.cat_id = c_bis.cat_id)))))
          AND iw.iw_prefix = iw_bis.iw_prefix)))) LIMIT 1

BETA_QnB = SELECT DISTINCT iw.iw_url FROM interwiki AS iw
       WHERE (iw.iw_prefix = 'fr'
       AND (NOT EXISTS
           (SELECT * FROM interwiki AS iw_bis
            WHERE ((iw_bis.iw_url <> iw.iw_url
               OR (NOT EXISTS
                     (SELECT * FROM category AS c
                      WHERE (NOT EXISTS
                         (SELECT * FROM category AS c_bis
                          WHERE ((NOT EXISTS
                             (SELECT * FROM externallinks AS el,
                                   categorylinks AS cl,
                                   page AS p
                              WHERE (iw_bis.iw_url = el.el_to
                                 AND el.el_from = cl.cl_from
                                 AND c_bis.cat_title = cl.cl_to
                                 AND el.el_from = p.page_id)))
                              AND c.cat_id = c_bis.cat_id))))))
            AND iw.iw_prefix = iw_bis.iw_prefix))))
```

Figure 4.2: SQL translations of queries $\rho_{Q_B}$ and $\beta_{Q_{nB}}$.

| | $\mathcal{W}_{0\%}$ | $\mathcal{W}_{0.001\%}$ | $\mathcal{W}_{0.01\%}$ | $\mathcal{W}_{0.1\%}$ | $\mathcal{W}_{1\%}$ | $\mathcal{W}_{2\%}$ |
|---|---|---|---|---|---|---|
| $Q_B$ | 1.15 | 1.149 | 1.151 | 1.151 | 1.151 | 1.149 |
| $\varphi_{Q_B}$ | 4.658 | 3.156 | 3.155 | 4.157 | 3.656 | 4.66 |
| $\beta_{Q_B}$ | 2.652 | 3.155 | 2.155 | 3.157 | 2.148 | 3.156 |
| $\rho_{Q_B}$ | 3.158 | 3.155 | 3.155 | 3.156 | 2.163 | 3.156 |
| $Q_{nB}$ | 5.659 | 6.159 | 6.159 | 5.658 | 5.659 | 6.153 |
| $\varphi_{Q_{nB}}$ | 4.659 | 4.659 | 4.658 | 4.661 | 4.661 | 4.66 |
| $\beta_{Q_{nB}}$ | 3.155 | 3.155 | 3.154 | 3.156 | 3.153 | 3.155 |
| $\rho_{Q_{nB}}$ | 3.659 | 3.154 | 3.155 | 3.152 | 3.155 | 3.656 |

Table 4.3: Execution times for the first experiment. Time is in milliseconds.

2. For the Boolean query, the certain SQL rewriting is at worst four times slower than the original query;

3. For the nonBoolean query, the certain SQL rewriting is always slightly faster than the original query. A possible explanation is that the additional conditions occurring in the subqueries of the certain SQL rewritings allow the DBMS to avoid some Cartesian products;

4. The execution times are independent of the amount of conflicting tuples in the database. Figures 4.3 and 4.4 illustrate the evolution of the evaluation times for queries $Q_B$ and $Q_{nB}$ and their certain SQL rewritings for a proportion of conflicting tuples that ranges from 0% to 4%.

This last observation was confirmed by some more experiments we conducted on various synthetic databases in which we obtained a similar result: the proportion of tuples involved in primary key violations does not affect the evaluation times. As the proportion of uncertainties in the database does not affect the resulting evaluation times, we only consider databases that are consistent in our second experiment.

### 4.2.2 Query and Data Complexities

The second experiment measures the impact of the number of tuples in the databases and of the number of atoms in the conjunctive query. We also study whether execution times depend on the quantifier rank, the quantifier block rank or the quantifier block number of the query. Unlike the databases of the previous experiment, databases in this second experiment are synthetic and consistent (that is, they contain no primary key violation).

A synthetic database that is consistent is easier to characterize than an inconsistent one: the parameters are mainly the number of tuples per relation and how those tuples join together. It allows us to choose queries that are more specific and more relevant to study the effect of the syntactic optimizations. Moreover, on a consistent database, the answer of the conjunctive query is the same as the answer of its certain first-order rewritings.

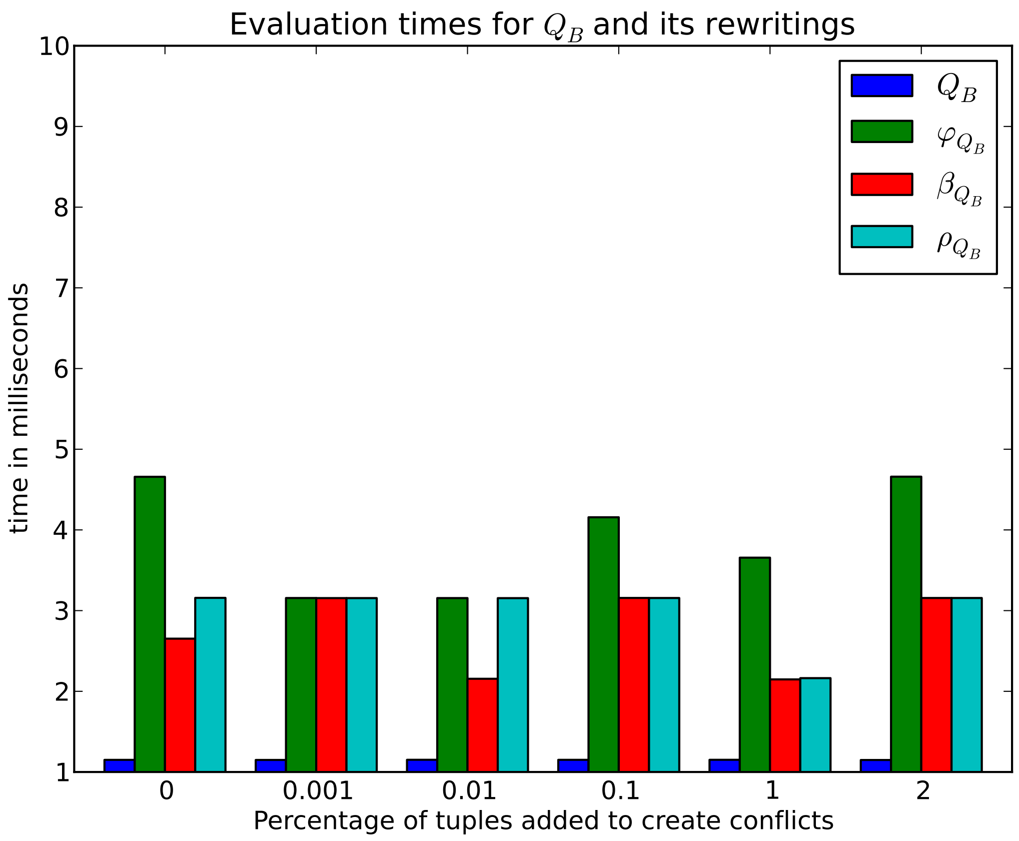

Figure 4.3: Time needed to evaluate $Q_B$ and its three certain SQL rewritings. The time is in milliseconds. The percentage of added tuples to create conflict ranges from 0% to 2%.

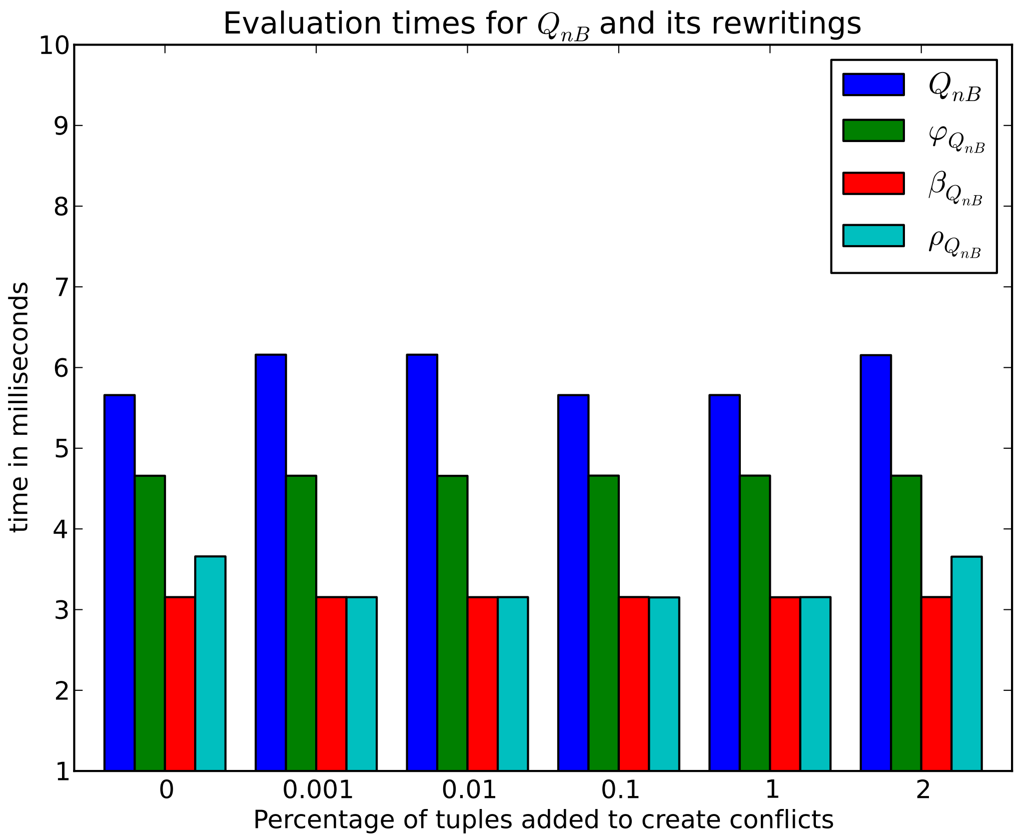

Figure 4.4: Time needed to evaluate $Q_{nB}$ and its three certain SQL rewritings. The time is in milliseconds. The percentage of added tuples to create conflict ranges from 0% to 2%.

### Databases and Queries

Consider schema $R_1, \ldots, R_{10}$ where each $R_i$ has signature $[2, 1]$. For $N$ in $\{10^2, 10^3, 5 \times 10^3, 10^4, 5 \times 10^4, 10^5\}$, we constructed a database denoted $\mathsf{db}_N$. Database $\mathsf{db}_N$ contains facts $R_i(1, 1), \ldots, R_i(N-1, N-1)$ and $R_i(N, b)$ for $1 \leq i \leq 10$, where $b$ is a constant. Hence, $\mathsf{db}_N$ contains $10 \times N$ tuples.

The queries that are used for this experiment are $\llbracket 1 \rrbracket, \ldots, \llbracket 10 \rrbracket$. Remember $\llbracket m \rrbracket = (\lfloor m \rfloor, \emptyset)$ where $\lfloor m \rfloor = \{R_1(\underline{x_1}, b), \ldots, R_m(\underline{x_m}, b)\}$. This query evaluates to true on each of the considered databases.

For each $1 \leq m \leq 10$, $\varphi_{\llbracket m \rrbracket}$ denotes the certain first-order rewriting obtained using `NaiveFo`, $\beta_{\llbracket m \rrbracket}$ denotes the certain first-order rewriting obtained using `GroupingFo` and $\rho_{\llbracket m \rrbracket}$ denotes the certain first-order rewriting obtained using `SplitFo`.

The following table recalls the number of quantifier blocks, the quantifier block rank and the quantifier rank of the certain first-order rewritings of $\llbracket m \rrbracket$. SQL translations of $\llbracket 3 \rrbracket, \varphi_{\llbracket 3 \rrbracket}, \beta_{\llbracket 3 \rrbracket}$ and $\rho_{\llbracket 3 \rrbracket}$ are shown in Figure 3.1.

| | $\varphi_{\llbracket m \rrbracket}$ | $\beta_{\llbracket m \rrbracket}$ | $\rho_{\llbracket m \rrbracket}$ |
|---|---|---|---|
| $\mathsf{qbn}(\cdot)$ | $2m$ | 2 | $2m$ |
| $\mathsf{qbr}(\cdot)$ | $2m$ | 2 | 2 |
| $\mathsf{qr}(\cdot)$ | $2m$ | $2m$ | 2 |

### Observations and Measurements

The 40 queries of this experiment are evaluated on the six different databases. Table 4.4 summarizes the results. The evaluation times vary from 0.6 milliseconds (for the smallest conjunctive query on the smallest database) to 378 milliseconds for $\rho_{\llbracket 10 \rrbracket}$ on the biggest database. We make the following observations:

1. The certain SQL rewriting is always less than two times slower than the conjunctive query;

2. Figure 4.5 illustrates the behaviour of the evaluation times for $\llbracket 10 \rrbracket$ and for its three rewritings when the number of tuples in the database increases, from $1,000$ to $1,000,000$ tuples. Although the execution times are clearly dependent of the number of tuples in the database, the evolution of the execution times for the certain SQL rewritings is similar to the evolution of the execution times for the conjunctive query with respect to the number of tuples in the database;

3. A similar observation can be made with respect of the number of atoms in the conjunctive query. Figure 4.6 shows the execution times of $\llbracket m \rrbracket$ and its certain SQL rewritings for $1 \leq m \leq 10$ on $\mathsf{db}_{10,000}$;

4. It can be seen in Figures 4.5 and 4.6 that the curve of $\beta_{\llbracket m \rrbracket}$ is below the curve of $\varphi_{\llbracket m \rrbracket}$. Query $\beta_{\llbracket m \rrbracket}$ lowers the execution times of $\varphi_{\llbracket m \rrbracket}$ with about 20% on average;

5. Notice that the execution time of $\rho_{\llbracket m \rrbracket}$ is sometimes higher than the execution time of $\varphi_{\llbracket m \rrbracket}$ (see $m = 10$ on $\mathsf{db}_{100,000}$ for instance). We cannot explain this behavior, we think the small variations are induced by some side effects in the DBMS during the process or a different query plan chosen and executed by the DBMS.

### 4.2.3 Conclusions

We conducted two sets of experiments. The first experiment focused on a large database of practical interest while the second experiment considered synthetic, consistent databases.

The resulting execution times provide a strong confidence that the certain first-order rewriting approach is an efficient way to deal with uncertainty in databases if the query is an acyclic **SJFC** query whose attack graph is acyclic. The first experiment showed that the execution times for the certain SQL rewritings are close to the execution times of the original conjunctive queries. This experiment also showed that the execution times are not affected by the proportion of tuples that are involved in a primary key violation.

In the second experiment, we measured the impact of the size of the database, of the size of the query, and of the two considered syntactic optimizations. Again, we observed acceptable execution times for the certain SQL rewritings: the execution times are always less than two times those of the conjunctive query. The increase of the size of the database or the increase of the size of the query seems to result in a linear increase of the execution times of the certain SQL rewritings. We observed that the syntactic optimization that reduces the number of quantifier blocks (i.e. function `GroupingFo`) results in a certain SQL rewriting that runs slightly faster than the "naive" certain SQL rewriting. The performance gain can reach 30%. It is on average of 15% but is not systematic.

The reduction of the number of quantifier blocks, of the quantifier block rank or of the quantifier rank is not the only way to reduce the execution time of the SQL translations of certain first-order rewritings. The following example suggests that parameters other than $\mathsf{qbn}(\cdot), \mathsf{qbr}(\cdot)$ and $\mathsf{qr}(\cdot)$ play an important role.

**Example 4.5** Consider the following query $q = \exists y\big(R(\underline{x}, y) \wedge S(\underline{y, z})\big)$. Query $q$ has a certain first-order rewriting, namely:

$$\varphi = \exists y\Big(R(\underline{x}, y) \wedge \forall y'\big(R(\underline{x}, y') \rightarrow S(\underline{y', z})\big)\Big).$$

There are at least two certain SQL rewritings for $q$. The following SQL queries `Q1` and `Q2` are two examples of such rewritings.

```
Q1 = SELECT s1.B, r1.A
     FROM   S as s1, R as r1
     WHERE  s1.A = r1.B
     AND    NOT EXISTS ( SELECT *
                         FROM   R as r2
                         WHERE  r2.A = r1.A
                         AND    NOT EXISTS ( SELECT *
                                             FROM   S as s2
                                             WHERE  s2.B = s1.B
                                             AND    s2.A = r2.B ) )


Q2 = SELECT s1.B, r1.A
     FROM   S as s1, R as r1
```

```
WHERE  NOT EXISTS ( SELECT *
                    FROM   R as r2
                    WHERE  r2.A = r1.A
                    AND    NOT EXISTS ( SELECT *
                                        FROM   S as s2
                                        WHERE  s2.B = s1.B
                                        AND    s2.A = r2.B ) )
```

The only difference between `Q1` and `Q2` is an additional condition `s1.A = r1.B` in the outermost `WHERE` in `Q1`. Intuitively, `Q1` is a translation of $q \wedge \varphi$ while `Q2` is a translation of $\varphi$ in SQL. As $\varphi \rightarrow q$, it is easy to see that `Q1` and `Q2` are semantically equivalent. The execution time heavily depends on the translation chosen, as it is shown in the following table. This table gives the time to get the answer of those queries on two consistent databases of $1,000$ tuples and $10,000$ tuples.

| number of tuples | SQL of $q$ | `Q1` | `Q2` |
|---|---|---|---|
| $1,000$ | 11.4 ms | 11.5 ms | 44.21 ms |
| $10,000$ | 69.25 ms | 73.74 ms | $> 30,000$ ms |

The syntactic difference, which is clearly not related to the number of quantifier blocks, the quantifier block rank or the quantifier rank, seems to have a big impact on the execution times. Even on a small database of $1,000$ tuples, the execution time of `Q2` is 4 times higher than the one of `Q1`. It takes more than 30 seconds to get the answer of query `Q2` on a database of $10,000$ tuples. $\triangleleft$

Although certain first-order rewritings obtained using our functions generally have execution times that are comparable to the ones of the conjunctive queries, this last example suggests that the performance can depend on some factors that are unrelated to the number of quantifier blocks, to the quantifier block rank or to the quantifier rank. Currently, we have not been able to identify these factors.

| | $\mathsf{db}_{100}$ | $\mathsf{db}_{1,000}$ | $\mathsf{db}_{5,000}$ | $\mathsf{db}_{10,000}$ | $\mathsf{db}_{50,000}$ | $\mathsf{db}_{100,000}$ |
|---|---|---|---|---|---|---|
| $\llbracket 1 \rrbracket$ | 0.648 | 2.152 | 3.147 | 4.148 | 16.678 | 31.697 |
| $\varphi_{\llbracket 1 \rrbracket}$ | 0.648 | 2.152 | 3.655 | 4.154 | 18.171 | 34.71 |
| $\beta_{\llbracket 1 \rrbracket}$ | 0.647 | 2.152 | 3.153 | 4.648 | 18.178 | 39.212 |
| $\rho_{\llbracket 1 \rrbracket}$ | 0.647 | 2.152 | 3.147 | 5.152 | 17.677 | 34.707 |
| $\llbracket 2 \rrbracket$ | 1.149 | 2.652 | 4.148 | 6.15 | 29.695 | 67.77 |
| $\varphi_{\llbracket 2 \rrbracket}$ | 1.149 | 3.656 | 5.156 | 8.154 | 35.705 | 72.279 |
| $\beta_{\llbracket 2 \rrbracket}$ | 1.149 | 3.155 | 4.155 | 7.154 | 29.694 | 72.78 |
| $\rho_{\llbracket 2 \rrbracket}$ | 1.15 | 3.154 | 5.155 | 8.656 | 35.204 | 79.79 |
| $\llbracket 3 \rrbracket$ | 1.149 | 3.657 | 6.158 | 9.164 | 47.73 | 96.834 |
| $\varphi_{\llbracket 3 \rrbracket}$ | 1.651 | 5.16 | 6.66 | 11.667 | 54.74 | 125.874 |
| $\beta_{\llbracket 3 \rrbracket}$ | 1.149 | 4.157 | 5.659 | 9.156 | 48.732 | 102.838 |
| $\rho_{\llbracket 3 \rrbracket}$ | 1.651 | 5.161 | 7.66 | 11.662 | 51.736 | 110.854 |
| $\llbracket 4 \rrbracket$ | 1.149 | 4.659 | 6.659 | 11.166 | 64.262 | 120.871 |
| $\varphi_{\llbracket 4 \rrbracket}$ | 1.651 | 3.656 | 9.656 | 14.67 | 68.764 | 164.959 |
| $\beta_{\llbracket 4 \rrbracket}$ | 1.651 | 5.161 | 6.659 | 12.162 | 64.767 | 116.861 |
| $\rho_{\llbracket 4 \rrbracket}$ | 1.651 | 5.16 | 8.161 | 14.674 | 78.787 | 148.913 |
| $\llbracket 5 \rrbracket$ | 1.149 | 5.159 | 7.152 | 13.665 | 75.29 | 140.903 |
| $\varphi_{\llbracket 5 \rrbracket}$ | 2.152 | 3.154 | 10.157 | 18.178 | 85.798 | 189.001 |
| $\beta_{\llbracket 5 \rrbracket}$ | 1.65 | 3.155 | 9.665 | 14.666 | 73.777 | 145.411 |
| $\rho_{\llbracket 5 \rrbracket}$ | 2.152 | 3.147 | 9.655 | 18.172 | 85.295 | 168.437 |
| $\llbracket 6 \rrbracket$ | 1.65 | 2.651 | 8.665 | 16.17 | 90.813 | 179.481 |
| $\varphi_{\llbracket 6 \rrbracket}$ | 2.653 | 3.646 | 11.66 | 21.678 | 112.353 | 210.027 |
| $\beta_{\llbracket 6 \rrbracket}$ | 2.154 | 3.154 | 9.659 | 17.174 | 90.302 | 169.444 |
| $\rho_{\llbracket 6 \rrbracket}$ | 2.652 | 3.147 | 11.659 | 21.679 | 104.331 | 230.574 |
| $\llbracket 7 \rrbracket$ | 1.652 | 3.152 | 9.662 | 18.678 | 101.332 | 207.532 |
| $\varphi_{\llbracket 7 \rrbracket}$ | 3.172 | 4.154 | 13.664 | 25.187 | 120.36 | 261.108 |
| $\beta_{\llbracket 7 \rrbracket}$ | 2.153 | 3.148 | 10.665 | 21.686 | 104.338 | 224.576 |
| $\rho_{\llbracket 7 \rrbracket}$ | 2.654 | 4.148 | 14.668 | 25.183 | 119.364 | 256.123 |
| $\llbracket 8 \rrbracket$ | 1.65 | 3.148 | 11.166 | 21.681 | 114.357 | 217.044 |
| $\varphi_{\llbracket 8 \rrbracket}$ | 3.155 | 4.154 | 15.167 | 28.698 | 145.905 | 307.206 |
| $\beta_{\llbracket 8 \rrbracket}$ | 2.151 | 3.152 | 12.162 | 22.69 | 128.892 | 256.63 |
| $\rho_{\llbracket 8 \rrbracket}$ | 3.155 | 4.155 | 15.166 | 28.699 | 147.922 | 272.639 |
| $\llbracket 9 \rrbracket$ | 2.153 | 3.65 | 12.669 | 26.19 | 127.89 | 237.584 |
| $\varphi_{\llbracket 9 \rrbracket}$ | 3.656 | 5.158 | 17.673 | 31.7 | 169.956 | 346.284 |
| $\beta_{\llbracket 9 \rrbracket}$ | 2.653 | 3.655 | 13.17 | 25.689 | 124.877 | 252.613 |
| $\rho_{\llbracket 9 \rrbracket}$ | 3.656 | 4.66 | 16.67 | 31.704 | 152.419 | 348.795 |
| $\llbracket 10 \rrbracket$ | 2.152 | 3.647 | 13.163 | 25.688 | 134.391 | 266.233 |
| $\varphi_{\llbracket 10 \rrbracket}$ | 3.656 | 5.155 | 19.176 | 35.206 | 191.501 | 336.26 |
| $\beta_{\llbracket 10 \rrbracket}$ | 2.654 | 4.147 | 14.668 | 29.203 | 147.923 | 310.731 |
| $\rho_{\llbracket 10 \rrbracket}$ | 3.657 | 4.649 | 18.673 | 38.721 | 184.982 | 378.849 |

Table 4.4: Execution times for the second experiment. Time is in milliseconds.

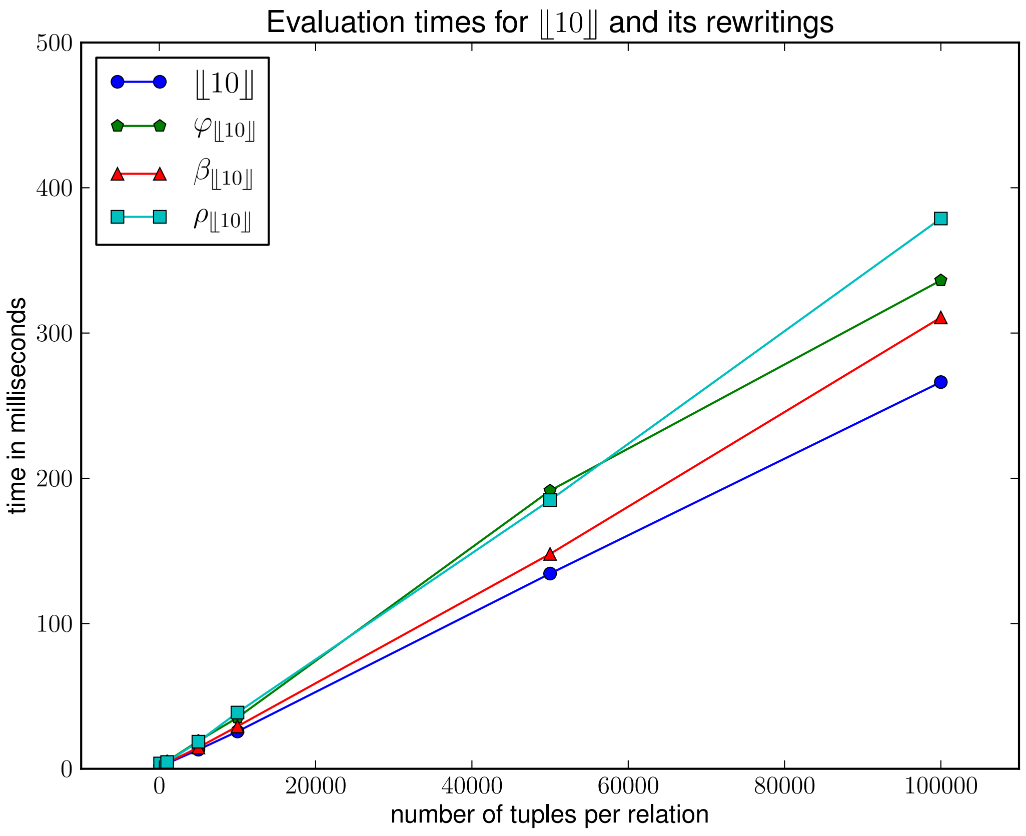

Figure 4.5: Time needed to evaluate $[\![10]\!]$ and its rewritings. The time is in milliseconds. The number of tuples per relation ranges from 100 to $100,000$.

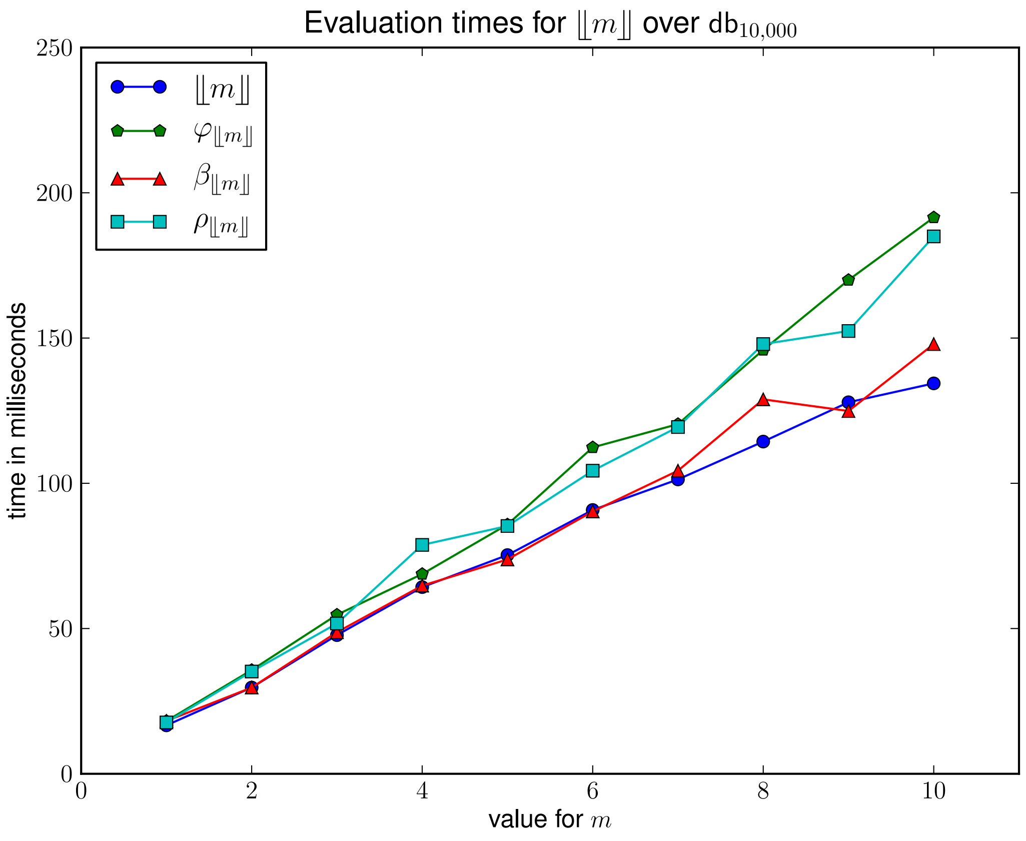

Figure 4.6: Time needed to evaluate $\llbracket m \rrbracket$ and its rewritings for $1 \leq m \leq 10$. Time is in milliseconds. The database is fixed and contains $10,000$ tuples per relation.

CHAPTER 5

# Conclusions

In the first part of the thesis, we dealt with certain query answering on relational databases that are allowed to violate primary key constraints. We say that a Boolean query $q$ is certain in such an uncertain database $\mathsf{db}$ if $q$ evaluates to true on every repair of $\mathsf{db}$. Given a Boolean query $q$, $\mathsf{CERTAINTY}(q)$ is defined as the set of uncertain databases in which $q$ is certain. We focused on the case where $\mathsf{CERTAINTY}(q)$ is first-order definable.

The set $\mathsf{CERTAINTY}(q)$ is first-order definable if and only if there exists a first-order sentence $\varphi$ such that, for every database $\mathsf{db}$, $\varphi$ is true on $\mathsf{db}$ if and only if $q$ is certain in $\mathsf{db}$. Such sentence $\varphi$ is called a certain first-order rewriting for $q$.

In [Wijsen 2010], it was shown that if $q$ is an acyclic **SJFC** query, then the first-order definability of $\mathsf{CERTAINTY}(q)$ is decidable. This characterization relies on the attack graph of $q$. A certain first-order rewriting for $q$ exists if and only if the attack graph of $q$ is acyclic. We provided a function that, given a query $q$ with an acyclic attack graph, produces a certain first-order rewriting for $q$.

We showed that the SQL translation of the certain first-order rewritings produced by this function can contain deeply nested subqueries. We studied the number of quantifier blocks, the quantifier block rank, and the quantifier rank of certain first-order rewritings. We presented two syntactic optimizations that aim to reduce the number of (nested) subqueries in the SQL translations of those certain first-order rewritings.

We then conducted experiments to investigate whether the certain first-order rewriting technique is an efficient way to deal with uncertainty in practical databases. We measured the execution times of several queries and their certain first-order rewritings. We found that certain first-order rewriting is a technique that scales well, both for data complexity and query complexity. This approach also scales well when the proportion of tuples involved in primary key violations increases.

Our experimental results suggest that the approach of computing the certain answer to a query using the certain first-order rewriting technique is efficient, practical and scalable. The two considered syntactic optimizations lead to a performance improvement. The gain is on average of 15% but is not systematic.

There are still several open questions about $\mathsf{CERTAINTY}(q)$. Relatively little is known in the case of conjunctive queries $q$ that are cyclic and/or contain self-joins. It is also an open conjecture that, for every conjunctive query $q$ without self-join, $\mathsf{CERTAINTY}(q)$ is either in **P** or **coNP**-complete. This dichotomy has been proven to be true for queries with exactly two atoms [Kolaitis and Pema 2012]. [Fontaine 2013] explains why it seems hard to get such a dichotomy for consistent query answering: it would imply a solution for the dichotomy conjecture for the constraint satisfaction problem, a famous long-standing open problem.

# Part II

# A Pattern Matching Problem for Multiwords

CHAPTER 6

# A Variant of the Pattern Matching Problem

The second part of the thesis deals with uncertainty in the framework of first- order logic on words. In this setting, uncertainty is captured by the concept of multiword, which is a finite sequence of nonempty sets of possible symbols. Every word obtained by selecting one symbol from each set of possible symbols is a possible word. In particular, we are interested in the following variant of the pattern matching problem: given a word $w$, is $w$ a factor of every possible word of a given multiword?

Section 6.1 situates our pattern matching problem using multiwords among several other variants of the pattern matching problem.

In Section 6.2, we recall the notions of words and partial words. Partial words are words in which don't-care symbols are allowed and can be replaced by any symbol of the alphabet. Partial words are a special case of multiwords. Given a word $w$, we define $\mathsf{CERTAIN}(w)$ as the set of multiwords such that $w$ is a factor of every possible word of the multiword. We postulate the conjecture that $\mathsf{CERTAIN}(w)$ is first-order expressible for every word $w$. In a forthcoming chapter, we will show the first-order definability of $\mathsf{CERTAIN}(w)$ under rather weak restrictions on $w$.

The variant of the pattern matching problem with multiwords has a direct application in the context of certain query answering in uncertain database history. In Section 6.3, we introduce the notion of database histories and uncertainty by primary key violations in database histories. We show how the problem of certain query answering in uncertain database histories can be viewed as an application of the variant of the pattern matching problem with multiwords.

## 6.1 Words with don't-care Symbols

Given a pattern $w$ and a text $t$, the *pattern matching problem* is to find all the occurrences of the word $w$ in $t$. There exist efficient algorithms that solve this problem, like the well-known Knuth-Morris-Pratt algorithm [Knuth et al. 1977] and Boyer-Moore algorithm [Boyer and Moore 1977] (Chapters 3 and 4 in [Crochemore and Rytter 1994]).

Several extensions of this problem have been studied. Instead of a single pattern $w$, the Aho-Corasick algorithm efficiently finds in a text $t$ all the occurrences of words $w$ taken from a finite set of words [Aho and Corasick 1975]. A more general problem is the regular expression matching problem where the pattern is a set of words specified by a regular expression (see for instance Chapter 7 in [Crochemore and Rytter 1994]).

Other extensions deal with the pattern matching problem by allowing *don't-care* symbols in the pattern $w$ and/or in the text $t$. In this case, some positions in the pattern or in the text can contain a set of symbols, instead of a single symbol. A word with don't-care symbols represents a finite set of (classical) words obtained by selecting a single symbol among the symbols provided in each don't-care position. If $w$ is a pattern with don't-care symbols and $t$ is a text, the problem consists in finding all the occurrences of words represented by $w$ in the text $t$. When $w$ is a pattern and $t$ is a text with don't-care symbols, we are interested in finding the occurrences of $w$ in $t$ such that in each don't-care position $i$, the symbol at the corresponding position of $w$ belongs to the set of symbols of $t$ at position $i$.

When don't-care symbols are allowed, most of the existing exact methods for pattern matching are useless or have to be adapted. One among the first works in this framework has been presented by Fisher and Paterson in [Fischer and Paterson 1974]. Without being exhaustive, let us also mention the recent references [Holub et al. 2008; Rahman et al. 2007; Kucherov et al. 2007].

The interest in words with don't-care symbols is driven by applications in computational biology, cryptanalysis, musicology, and other areas. In computational biology, DNA sequences may still be considered to match each other if letter A (respectively, C) is juxtaposed with letter T (respectively, G); analogous juxtapositions may count as matches in protein sequences. In cryptanalysis, so far undecoded symbols may be known to match one of a specific set of letters in the alphabet. In music, single notes may match chords, or notes separated by an octave.

In the literature, different terms have been used for words with don't-care symbols like indeterminate words [Holub et al. 2008], partial words, words with holes or jokers [Berstel and Boasson 1999; Blanchet-Sadri 2007; Crochemore et al. 2007]. In each case, either the don't-care symbol means any symbol of the alphabet, or it has to be selected among a subset of the alphabet depending on its position in the word. We follow the second approach and we use the term *multiword.* The notion of partial word has been generalized in [Halava et al. 2007] by the concept of relational word. We will use the term partial word to refer to words where don't-care positions represent the entire alphabet.

Uncertainty in words is captured by the concept of multiwords. A multiword is a finite sequence of nonempty sets of possible symbols. For example, multiword $M = \langle\{a, b\}, \{a\}, \{b\}, \{b\}\rangle$ encodes a word $m$ in which the first symbol is a don't-care symbol and the symbols at the three last positions are $a$, $b$ and $b$. The set $\{a, b\}$ at the first position in $M$ means that we do not know exactly what is the first symbol of $m$ but it has to be $a$ or $b$. A multiword gives rise to a finite set of possible words that are obtained by selecting a single symbol in each set of symbols of the multiword. The possible words of $M$ are $aabb$ and $babb$.

We are interested in the following variant of the pattern matching problem: given a word $w$, is $w$ certain in a given multiword, meaning that $w$ is a factor of every possible word of the multiword? For example, $ab$ and $bb$ are certain in $M$, but $aa$ is not, because $aa$ is not a factor of the possible word $babb$.

## 6.2 Definitions and Preliminaries

**Definition 6.1** (Words) Let $\Sigma$ be a finite set. We call $\Sigma = \{a, b, c, \dots\}$ an *alphabet* and elements of $\Sigma$ are called *symbols* or *letters*. A *word* of *length* $n \geq 0$ *over* $\Sigma$ is a total function $w$: $\{1, \dots, n\} \to \Sigma$. As usual, we write such a word $w_1 \cdots w_n$, where $w_i = w(i)$ is the symbol at *position* $i$. The *empty word*, denoted by $\epsilon$, has length 0. The length of a word $w$ is denoted by $|w|$.

The set $\Sigma^*$ contains every word over $\Sigma$. The set $\Sigma^+$ contains every nonempty word over $\Sigma$. A *language over* $\Sigma$ (or simply language if there is no ambiguity) is a possibly empty subset of $\Sigma^*$. If $L$ is a language over $\Sigma$, we denote by $\overline{L}$ the set $\Sigma^* \setminus L$.

If $u$ and $v$ are two words over $\Sigma$, then the concatenation of words $u$ and $v$ is denoted $uv$ or $u \cdot v$ and is also a word over $\Sigma$. Let $w \in \Sigma^+$, then $w^1 = w$ and, for $k > 1$, $w^k = w \cdot w^{k-1}$. If $w = pq$ for some words $p$ and $q$, then $p$ is called a *prefix* of $w$ and $q$ is called a *suffix* of $w$. A prefix or a suffix of $w$ that is different from $w$ is called *proper*. We say that a word $w$ is a *factor* of $v$, denoted by $v \Vdash w$, if there exist words $p$ and $q$ such that $v = pwq$. $\lhd$

**Definition 6.2** (Multiwords) We define the *powerset alphabet* as $\widehat{\Sigma} = 2^{\Sigma} \setminus \{\emptyset\}$. A *multiword* $M = A_1 A_2 \cdots A_n$ is a finite word over the powerset alphabet $\widehat{\Sigma}$, i.e. $A_i \subseteq \Sigma$ and $A_i \neq \emptyset$ for all $i$.

Given a multiword $M = A_1 A_2 \cdots A_n$, we define the set of words represented by $M$:

$$\mathsf{words}(M) := \{a_1 a_2 \cdots a_n \mid \forall i \in \{1, \dots, n\} : a_i \in A_i\}.$$

We say that a word $w$ is *certain in* $M$, denoted $M \models_{\mathsf{certain}} w$, if $w$ is a factor of every word in $\mathsf{words}(M)$. $\lhd$

Given a nonemtpy word $w$, the set $\mathsf{CERTAIN}(w)$ defined next is the set of every multiword such that $w$ is certain in the multiword.

**Definition 6.3** Let $w \in \Sigma^+$ be a word. The set $\mathsf{CERTAIN}(w)$ is defined as follows.

$$\mathsf{CERTAIN}(w) := \{M \in \widehat{\Sigma}^* \mid M \models_{\mathsf{certain}} w\}.$$

$\lhd$

**Example 6.1** Let $M = abdabca\{a,b\}bdab\{c,d\}abcab$, a multiword. Curly braces are omitted for symbols that are singletons; for example, $\{a\}$ is written as $a$. There are two don't-care positions in $M$ with sets $\{a,b\}$ and $\{c,d\}$. We have:

$$\begin{aligned}\mathsf{words}(M) \;=\{ \;\; & abdabca\underline{\mathbf{a}bdab\mathbf{c}ab}cab,\\ & abdabca\mathbf{a}bd\underline{ab\mathbf{d}abcab},\\ & \underline{abdabca\mathbf{b}}bdab\mathbf{c}abcab,\\ & \underline{abdabca\mathbf{b}}bd\underline{ab\mathbf{d}abcab} \;\; \}.\end{aligned}$$

Hence, $M \models_{\mathsf{certain}} abdabcab$ because $abdabcab$ is a factor (underlined for readability) of each word in $\mathsf{words}(M)$. We have $M \in \mathsf{CERTAIN}(abdabcab)$. $\triangleleft$

**Definition 6.4** A *partial word* of length $n$ is a multiword $M = A_1A_2\cdots A_n$ where, for each $i$, either $A_i = \Sigma$, or $A_i = \{a\}$. The term "partial" refers to the fact that a partial word of length $n$ over $\Sigma$ is defined by a partial function $w : \{1,\ldots,n\} \to \Sigma$. Every partial word can be extended into a multiword by assuming $A_i = \Sigma$ if $w$ is undefined in position $i$. In a partial word $M = A_1A_2\ldots A_n$, it is common to denote the letter $\Sigma$ by $\diamond$ [Berstel and Boasson 1999]. $\triangleleft$

**Example 6.2** Let $\Sigma = \{a,b,c\}$. The partial word $M_\diamond = a\{a,b,c\}b\{a,b,c\}c$ is also denoted $a\diamond b\diamond c$ following notation in [Berstel and Boasson 1999]. The set of possible words of $M_\diamond$ is:

$$\mathsf{words}(M_\diamond) = \{aabac, aabbc, aabcc, abbac, abbbc, abbcc, acbac, acbbc, acbcc\}$$

$\triangleleft$

Finally, we define $\mathsf{CERTAIN}_\diamond(w)$ as the restriction of $\mathsf{CERTAIN}(w)$ to partial words:

$$\mathsf{CERTAIN}_\diamond(w) = \{M \mid M \text{ is a partial word and } M \models_{\mathsf{certain}} w\}$$

Note that, over an alphabet $\Sigma$ of exactly two symbols, every multiword is a partial word.

A word over a finite alphabet can be represented as first-order structure [Libkin 2004]. The following definition explains relationships between words and first-order structures, and between languages and first-order sentences.

**Definition 6.5** For a finite alphabet $\Sigma$, we define a first-order vocabulary, denoted $\sigma_\Sigma$, which contains the following predicate symbols:

1. $\sigma_\Sigma$ contains the binary predicates $<$ and $S$ (successor) ;
2. for every $a \in \Sigma$, $\sigma_\Sigma$ contains a unary predicate $P_a$.

Every word $w$ of length $n$ defines a $\sigma_\Sigma$-structure $\mathsf{struct}(w)$ as follows.

1. The universe of $\mathsf{struct}(w)$ is $\{1,2,\ldots,n\}$ ;

2. $S$ is interpreted by $\{(1,2),(2,3),\dots(n-1,n)\}$ and $<$ by the transitive closure of $S$;
3. For every $a \in \Sigma$, $P_a$ is interpreted by the set of positions in $w$ where $a$ occurs.

A language $L$ is said to be *first-order definable* if there exists a first-order sentence $\varphi$ such that

$$L = \{w \in \Sigma^* \mid \mathsf{struct}(w) \models \varphi\}.$$

$\triangleleft$

**Example 6.3** Let $\Sigma = \{a, b\}$. Consider word $w = abaab$.
We have $\mathsf{struct}(w) = \langle \{1,2,3,4,5\}, <, S, P_a, P_b \rangle$, where $P_a = \{1,3,4\}$ and $P_b = \{2,5\}$. $\triangleleft$

Notice that every multiword is a word hence all definitions that apply on words also apply on multiwords.

**Example 6.4** A multiword $M$ belongs to $\mathsf{CERTAIN}(ab)$ if $\mathsf{struct}(M)$ satisfies the following formula.

$$\exists i \exists j \Big( (i < j) \wedge P_{\{a\}}(i) \wedge P_{\{b\}}(j) \wedge \forall k \big( (i < k < j) \rightarrow P_{\{a,b\}}(k) \big) \Big).$$

Intuitively, $ab$ is certain in a word if there is one position where "a" occurs, some greater position where "b" occurs and every position between those two positions contains either "a" or "b" (represented in a multiword by the set $\{a, b\}$ of possible symbols). $\triangleleft$

**Example 6.5** We will see in Section 8.8 that the word $aba$ is certain in a multiword $M$ if and only if $M$ contains the sequence $\langle \{a\}, \{b\}, \{a\} \rangle$. It follows that a multiword $M$ belongs to $\mathsf{CERTAIN}(aba)$ if $\mathsf{struct}(M)$ satisfies the following formula.

$$\exists i \exists j \exists k \big( S(i,j) \wedge S(j,k) \wedge P_{\{a\}}(i) \wedge P_{\{b\}}(j) \wedge P_{\{a\}}(k) \big).$$

$\triangleleft$

It follows from Examples 6.4 and 6.5 that $\mathsf{CERTAIN}(ab)$ and $\mathsf{CERTAIN}(aba)$ are first-order definable.

Given a word $w$, we are interested in the first-order definability of $\mathsf{CERTAIN}(w)$. It is an open conjecture that $\mathsf{CERTAIN}(w)$ is first-order definable for every word $w$. We experimentally verified this conjecture for more than 80,000,000 words for several sizes of alphabets and several lengths of words. In Chapter 8, we prove this conjecture under some rather weak assumptions on $w$.

The first-order definability of $\mathsf{CERTAIN}(w)$ has direct applications in the context of certain query answering in database histories. We introduce in the next section an example of such an application for uncertain database histories, i.e. database histories that are allowed to violate primary key constraints. We show a link between the first-order definability of $\mathsf{CERTAIN}(w)$ and the computation of the certain answer of a Boolean Linear-Time First-Order Temporal query on an uncertain database history.

## 6.3 Application in Uncertain Database Histories

A database history is a finite sequence of snapshot databases sharing the same schema. We assume a discrete and linear time scale. Consecutive snapshots have consecutive associated time points. Consider for example the relation WorksFor which contains two attributes: Name and Company. Let Name be the primary key. An example of such a relation is given next (primary key is underlined).

| WorksFor | Name | Company |
|---|---|---|
| | Ed | IBM |
| | John | MS |

In this example, a tuple $\langle \text{Ed}, \text{IBM} \rangle$ means there exists an employee whose name is Ed and who is currently working at IBM. Database histories help us to store information about the evolution of an employee's company. Here is an example of such a database history with the relation WorksFor at four successive time points $t_0, t_1, t_2$ and $t_3$.

| WorksFor, $t_0$ | Name | Company |
|---|---|---|
| | Ed | IBM |
| | John | MS |

| WorksFor, $t_1$ | Name | Company |
|---|---|---|
| | Ed | IBM |
| | John | MS |

| WorksFor, $t_2$ | Name | Company |
|---|---|---|
| | Ed | IBM |
| | John | MS |

| WorksFor, $t_3$ | Name | Company |
|---|---|---|
| | Ed | MS |
| | John | MS |

In this example, Ed moved from IBM to MS at $t_3$. The query *"Did* MS *recruit an* IBM *employee?"* can be expressed in first-order temporal logic (see, for example, [Tuzhilin and Clifford 1990]) as follows. In this example, $\bigcirc$ means "at the next time point".

$$q = \exists x(\mathsf{WorksFor}(\underline{x}, \text{IBM}) \wedge \bigcirc\mathsf{WorksFor}(\underline{x}, \text{MS}))$$

This query $q$ evaluates to true on our example database because there exists an employee, Ed in this case, who worked for IBM at some time point and for MS at the next time point.

We have considered so far that the primary key Name is satisfied. We now consider that database histories are allowed to violate the primary key. Primary key violations bring uncertainty in database histories. Consider the following database history $\mathsf{DB}_{\mathsf{his}}$ where the primary key Name is violated at times $t_1$ and $t_2$:

| WorksFor, $t_0$ | Name | Company |
|---|---|---|
| | Ed | IBM |

| WorksFor, $t_1$ | Name | Company |
|---|---|---|
| | Ed | IBM |
| | Ed | MS |

| WorksFor, $t_2$ | Name | Company |
|---|---|---|
| | Ed | IBM |
| | Ed | MS |

| WorksFor, $t_3$ | Name | Company |
|---|---|---|
| | Ed | MS |

Tuples $\langle \text{Ed}, \text{IBM}\rangle$ and $\langle \text{Ed}, \text{MS}\rangle$ at $t_1$ and $t_2$ mean that Ed worked for MS or IBM but we do not know which one. In order to make this history consistent, we have to delete one of the two tuples at $t_1$ and one of the two tuples at $t_2$. Thus, we have four repairs for $\mathsf{DB_{his}}$. An example of such repair where we choose to delete $\langle \text{Ed}, \text{MS}\rangle$ twice is shown next.

| WorksFor, $t_0$ | Name | Company |
|---|---|---|
| | Ed | IBM |

| WorksFor, $t_1$ | Name | Company |
|---|---|---|
| | Ed | IBM |

| WorksFor, $t_2$ | Name | Company |
|---|---|---|
| | Ed | IBM |

| WorksFor, $t_3$ | Name | Company |
|---|---|---|
| | Ed | MS |

It can be easily checked that query $q$ evaluates to true on each of the four possible repairs : there always exists a time point where Ed worked for IBM such that at the next time point, Ed worked for MS. Note that this time point may differ for each repair. In the context of uncertain database histories, we say that $q$ is *certain* in $\mathsf{DB_{his}}$ because $q$ is true on every repair of $\mathsf{DB_{his}}$.

We could encode Ed's employment history with the following sequence of sets of values:

$$S = \langle\{\text{IBM}\}, \{\text{IBM}, \text{MS}\}, \{\text{IBM}, \text{MS}\}, \{\text{MS}\}\rangle.$$

By analogy with the notion of repairs, this sequence may represent four possible sequences, each one can be obtained by selecting one value in each set of values. The four sequences are:

- $s_1 = \langle \text{IBM}, \text{IBM}, \text{IBM}, \text{MS}\rangle$;
- $s_2 = \langle \text{IBM}, \text{IBM}, \text{MS}, \text{MS}\rangle$;
- $s_3 = \langle \text{IBM}, \text{MS}, \text{IBM}, \text{MS}\rangle$ and
- $s_4 = \langle \text{IBM}, \text{MS}, \text{MS}, \text{MS}\rangle$.

Asking if there exists some employee who moved from IBM to MS is equivalent to checking if the sequence $\langle \text{IBM}, \text{MS}\rangle$ is a subsequence of $s_1, s_2, s_3$ and $s_4$, this is, if $\langle \text{IBM}, \text{MS}\rangle$ is a subsequence of every sequence represented by $S$.

Assume a finite alphabet $\Sigma = \{a, b\}$. A multiword may represent an uncertain database history. For example, if we substitute IBM by $a \in \Sigma$ and MS by $b \in \Sigma$, the following multiword $M = \{a\}\{a, b\}\{a, b\}\{b\}$ encodes the WorksFor relation at its four successive time points. $M$ has four possible words: $aaab$, $aabb$, $abab$, and $abbb$ that are related to the repairs of the uncertain database history. It is easy to see that the word $ab$, which abstracts $q$, is a factor of every possible word of $M$, which abstracts $\mathsf{DB_{his}}$, and as $M \in \mathsf{CERTAIN}(ab)$, $q$ is certain on $\mathsf{DB_{his}}$.

The problem of computing the certain answer in uncertain database histories can be seen as a variant of pattern matching: given a pattern $w$ and a text $t$ with don't-care symbols, does $w$ appear as a factor of each word $z$ represented by $t$? It is important to notice that we want to be sure that $w$ appears in *each* $z$, and not in *some* $z$. In the context of database histories, the queries ask whether a sequence $w = a_1 \cdots a_n$ is encountered in every repair, i.e. whether in every repair we can find a sequence $t_1 \cdots t_n$ of successive time points such that $a_i$ holds at $t_1$, $a_2$ holds at $t_2$ and so on. The first-order definability of $\mathsf{CERTAIN}(w)$ gives us a preliminary insight in certain linear-time first-order query rewriting.

CHAPTER 7

# The Problem $\mathsf{CERTAIN}(w)$

Given a word $w$, $\mathsf{CERTAIN}(w)$ is the set of all multiwords in which $w$ is certain, meaning that $w$ is a factor of every possible word of the multiword.

It is an open conjecture that $\mathsf{CERTAIN}(w)$ is first-order definable for every word $w$. Section 7.1 formalizes the problems we are interested in. We show that $\mathsf{CERTAIN}(w)$ and $\mathsf{CERTAIN}_{\diamond}(w)$ are regular languages and hence, their first-order definability is equivalent to their aperiodicity [Schützenberger 1965; McNaughton and Papert 1971].

In Section 7.2, we give a procedure to decide $\mathsf{CERTAIN}(w)$. We show that this procedure has several similarities with the well known Knuth-Morris-Pratt pattern matching algorithm.

$$\forall X_a \forall X_b \left( \left( \begin{array}{ll} & \forall y(X_a(y) \vee X_b(y)) \\ \wedge & \neg\exists y(X_a(y) \wedge X_b(y)) \\ \wedge & \forall y(X_a(y) \rightarrow P_{\{a\}}(y) \vee P_{\{a,b\}}(y)) \\ \wedge & \forall y(X_b(y) \rightarrow P_{\{b\}}(y) \vee P_{\{a,b\}}(y)) \end{array} \right) \rightarrow \exists z_1 \exists z_2 \left( \begin{array}{ll} & S(z_1, z_2) \\ \wedge & X_a(z_1) \\ \wedge & X_b(z_2) \end{array} \right) \right)$$

Figure 7.1: MSO definition of CERTAIN$(ab)$ for alphabet $\Sigma = \{a, b\}$.

## 7.1 Problem Statement

Given a word $w$, we defined CERTAIN$(w)$ as the set of all multiwords in which $w$ is certain. We are interested in the first-order definability of CERTAIN$(w)$ and we postulate the conjecture that CERTAIN$(w)$ is first-order definable for every word $w$. This conjecture had been experimentally checked for a large number of words of various lengths and over alphabets of various sizes.

We make progress in proving the conjecture. We first show the regularity for both CERTAIN$(w)$ and CERTAIN$_\diamond(w)$ for every word $w$. The regularity of CERTAIN$(w)$ is obtained by exhibiting an automaton recognizing this language. The construction of such an automaton is given in Chapter 9 and the regularity of CERTAIN$(w)$ is then deduced in Theorem 9.2. We argue that CERTAIN$_\diamond(w)$ is also regular as follows. Regular languages are closed under intersection, and since CERTAIN$_\diamond(w)$ is the intersection of CERTAIN$(w)$ and the (regular) set of all partial words, CERTAIN$_\diamond(w)$ is also regular.

The regularity of CERTAIN$(w)$ can also be deduced by showing that CERTAIN$(w)$ can be defined by a formula in *monadic second order-logic* (MSO). The regularity of CERTAIN$(w)$ follows by Büchi's theorem [Büchi 1960].

An example of such an MSO formula is given for the word $ab$ over $\Sigma = \{a, b\}$ in Figure 7.1. For any multiword $M$ over $\widehat{\Sigma}$, $ab$ is certain in $M$ if struct$(M)$ satisfies this MSO formula. Let struct$(M)$ be the structure $\langle \{1, \ldots, n\}, <, S, P_{\{a\}}, P_{\{b\}}, P_{\{a,b\}} \rangle$, the monadic second-order variables $X_a$ and $X_b$ range over all subsets of the universe $\{1, \ldots, n\}$. The first two conjuncts of the premise express that $X_a \cup X_b = \{1, \ldots, n\}$ and $X_a \cap X_b = \emptyset$. Variables $X_a$ and $X_b$ encode all words represented by the multiword $M$: if $X_a(y)$ is true, then $a$ is chosen at position $y$, and if $X_b(y)$ is true, then $b$ is chosen at position $y$. The premise states that for each position in multiword $M$, either $a$ or $b$ has to be chosen; $a$ can only be chosen at position $y$ if $M$ contains $\{a\}$ or $\{a, b\}$ at position $y$; $b$ can only be chosen at position $y$ if $M$ contains $\{b\}$ or $\{a, b\}$ at $y$. Notice that the premise depends on the alphabet. The consequence states that two successive positions contain $a$ and $b$ in that order; the successor relation $S(z_1, z_2)$ means that position $z_2$ is immediately after $z_1$.

Remember that we are interested in the first-order definability of CERTAIN$(w)$ for every word $w$.

**Theorem 7.1** *([Schützenberger 1965; McNaughton and Papert 1971]) The following are equivalent for each regular language $L \subseteq \Sigma^*$:*

1. *$L$ is aperiodic.*

*2. L is first-order definable.*

*3. L is star-free, i.e. L can be expressed by a regular expression with no Kleene star.*

Hence, the first-order definability of $\mathsf{CERTAIN}(w)$ and $\mathsf{CERTAIN}_{\diamond}(w)$ is equivalent to their aperiodicity.

**Definition 7.1** A language $L$ is aperiodic if there exists an integer $k > 0$ such that

$$\forall p, u, q \in \Sigma^* \ (pu^k q \in L \iff pu^{k+1} q \in L).$$

$\lhd$

As before, any definition that applies to words also applies to multiwords. We have $\mathsf{CERTAIN}(w)$ is *aperiodic* if there exists $k > 0$ such that,

$$\forall P, U, Q \in \widehat{\Sigma}^*, PU^k Q \models_{\mathsf{certain}} w \iff PU^{k+1} Q \models_{\mathsf{certain}} w.$$

The proof of the aperiodicity of $\mathsf{CERTAIN}(w)$ can be restricted to only one implication instead of an equivalence as in Definition 7.1. This is stated by the following lemma.

**Lemma 7.1** *A regular language L is aperiodic if and only if there exists an integer $n > 0$ such that*

$$\forall p, u, q \in \Sigma^* \ (pu^n q \in L \implies pu^{n+1} q \in L). \tag{7.1}$$

Proof. The *only-if* part is obvious. We show the *if* part. We use the classical result that for any finite monoid $M$, there exists an integer $m$ such that for any $s \in M$, $s^m$ is an idempotent, that is $s^{2m} = s^m$ [Pin 1986]. By this result applied to the syntactic monoid of $L$, there exists an integer $m$ such that

$$\forall p, u, q \in \Sigma^* \ (pu^m q \in L \iff pu^{2m} q \in L). \tag{7.2}$$

Let $n$ be an integer such that (7.1) holds. Let $k \geq \mathsf{max}(n, m)$, and let $i, j$ be integers such that $k = im + j$ with $0 \leq j < m$. We have the following equivalences.

$$\begin{array}{cl}
pu^{2k-j} q \in L & \\
\iff & (\text{as } 2k - j = 2im + j) \\
p(u^i)^{2m} u^j q \in L & \\
\iff & (\text{by (7.2)}) \\
p(u^i)^m u^j q \in L & \\
\iff & (\text{as } k = im + j) \\
pu^k q \in L &
\end{array}$$

To show the *if* part, it suffices to show that $\forall p, u, q \in \Sigma^*(pu^{k+1} q \in L \implies pu^k q \in L)$. Assume $pu^{k+1} q \in L$. We show $pu^k q \in L$. By (7.1), we have $pu^{k+2} q \in L$. By repeated applications of the same argument, we have $pu^{2k-j} q \in L$. By the aforementioned equivalences, we have $pu^{k+1} q \in L \implies pu^k q \in L$. $\square$

In Chapter 8, we establish the aperiodicity of $\mathsf{CERTAIN}(w)$ for a large class of words.

## 7.2 Deciding Membership of CERTAIN($w$)

Let $w$ be a word. We give a procedure for deciding the membership of CERTAIN($w$). This procedure relies on a construction defined in Lemma 7.3. The correctness of the procedure is established in Theorem 7.2.

The procedure is based on an idea which is quite similar to the shift function of the Knuth-Morris-Pratt pattern matching algorithm [Knuth et al. 1977].

The Knuth-Morris-Pratt pattern matching algorithm finds all occurrences of one given word (the pattern) within another one (the text). The algorithm relies on the use of a sliding window and a shift function to limit the number of comparisons needed to find the occurrences of the pattern.

The sliding window delimits a factor of the text which has the same length as the pattern. At each step of the algorithm, a comparison is made between the symbols of the sliding window and the symbols of the pattern. If this comparison results in a match, then the next symbols are compared. If the comparison results in a mismatch, then a shift value is computed using the shift function and the sliding window is shifted according to this value.

The shift function is constructed based on the observation that the pattern embodies information that can be used to determine the position where the next match could occur. This allows the algorithm to bypass some comparisons and to run in an average complexity smaller or equal to $\mathcal{O}(n)$ where $n$ is the length of the text. The shift function associates with every nonempty string its longest proper suffix that is a prefix of the pattern.

Let $p$ be a prefix of the text. Let $w$ be the pattern. Let $i$ be the current position in $w$ such that $w_1 \cdots w_i$ is a suffix of $p$. Assume $a$ is the next symbol to be read in the text and $p \cdot a$ leads to a mismatch (i.e. $w_{i+1} \neq a$). The shift function keeps the longest suffix of $p \cdot a$ which is also a prefix of $w$. The idea of the shift function is best explained with an example.

**Example 7.1** [Knuth et al. 1977] In this example, we juxtapose the pattern on the first line (which corresponds to the sliding window) with the text on second line. The underlined symbol indicates the current position in the text that is considered for comparison with the corresponding symbol of the pattern.

Let *babcbabcabcaabcabcabcacabc* be the text and let *abcabcacab* be the pattern. The pattern is placed at the left end and the first symbol considered for comparison is the leftmost one:

$$\begin{array}{cccccccccccccccccccccccccc}
a & b & c & a & b & c & a & c & a & b & & & & & & & & & & & & & & & & \\
\underline{b} & a & b & c & b & a & b & c & a & b & c & a & a & b & c & a & b & c & a & b & c & a & c & a & b & c
\end{array}$$

Since $b$ does not match $a$, we shift the pattern to the right for the next candidate position.

$$\begin{array}{cccccccccccccccccccccccccc}
 & a & b & c & a & b & c & a & c & a & b & & & & & & & & & & & & & & & \\
b & \underline{a} & b & c & b & a & b & c & a & b & c & a & a & b & c & a & b & c & a & b & c & a & c & a & b & c
\end{array}$$

This time we have a match. In order to check if we have an occurrence of our pattern in the text at this position, we go on comparing the next symbols. The next symbols $b$

and $c$ match, then comes a mismatch:

$$\begin{array}{cccccccccccccccccccccccccc}
 & a & b & c & a & b & c & a & c & a & b & & & & & & & & & & & & & & & \\
b & a & b & c & \underline{b} & a & b & c & a & b & c & a & a & b & c & a & b & c & a & b & c & a & c & a & b & c
\end{array}$$

The fourth symbol of the pattern does not match the text but the three first ones matched the text, so we know that the last four considered symbols of the text are $abcx$ with $x \neq a$. As our pattern does not start with something that can match $abcx$ with $x \neq a$, we can shift the window four positions to the right. This leads to the following situation, with a mismatch on the eighth symbol of the pattern.

$$\begin{array}{cccccccccccccccccccccccccc}
 & & & & & a & b & c & a & b & c & a & c & a & b & & & & & & & & & & & \\
b & a & b & c & b & a & b & c & a & b & c & a & \underline{a} & b & c & a & b & c & a & b & c & a & c & a & b & c
\end{array}$$

The last eight symbols were $abcabcax$ with $x \neq c$. This time, there exists a prefix of $w$ that is a proper suffix of $abcabcax$ with $x \neq c$: $abcab$. This allows us to shift the window three positions to the right.

$$\begin{array}{cccccccccccccccccccccccccc}
 & & & & & & & & a & b & c & a & b & c & a & c & a & b & & & & & & & & \\
b & a & b & c & b & a & b & c & a & b & c & a & \underline{a} & b & c & a & b & c & a & b & c & a & c & a & b & c
\end{array}$$

Again, there is a mismatch. We move the pattern four places to the right. This new position produces a match. We continue the comparisons until we reach a position with a mismatch, on the eighth symbol of the pattern:

$$\begin{array}{cccccccccccccccccccccccccc}
 & & & & & & & & & & & & a & b & c & a & b & c & a & c & a & b & & & & \\
b & a & b & c & b & a & b & c & a & b & c & a & a & b & c & a & b & c & a & \underline{b} & c & a & c & a & b & c
\end{array}$$

The window is moved three more positions to the right, leading to a match and an full occurrence of the pattern.

$$\begin{array}{cccccccccccccccccccccccccc}
 & & & & & & & & & & & & & & & a & b & c & a & b & c & a & c & a & b & \\
b & a & b & c & b & a & b & c & a & b & c & a & a & b & c & a & b & c & a & b & c & a & c & a & \underline{b} & c
\end{array}$$

◁

We define an operator called sufpre$(\cdot,\cdot)$ which brings the idea of this shift function to multiwords.

**Definition 7.2** Let $w$ be a word. If $u$ is a word, then sufpre$(u, w)$ denotes the maximal suffix of $u$ that is a prefix of $w$. For a set $S$ of words, we define sufpre$(S, w) = \{$sufpre$(u, w) \mid u \in S\}$. ◁

**Example 7.2** For example, sufpre$(abcd, cde) = cd$ and sufpre$(ab, c) = \epsilon$. Let $S = \{ba, aab\}$, then sufpre$(S, ab) = \{a, ab\}$. ◁

Let $S$ be a set of words. We write $\lfloor S \rfloor$ for the smallest set of words satisfying: $\lfloor S \rfloor \subseteq S$ and $\lfloor S \rfloor$ contains a suffix of every word in $S$.

**Example 7.3** If $S = \{aa, ac, abc, bc, c\}$, then $\lfloor S \rfloor = \{c, aa\}$ because $aa$ is a suffix of $aa$ and $c$ is a suffix of $ac, abc, bc$ and $c$. ◁

**Lemma 7.2** *If* $\mathsf{sufpre}(u, w) = q$*, then* $\mathsf{sufpre}(u \cdot a, w) = \mathsf{sufpre}(q \cdot a, w)$.

Proof. Assume $\mathsf{sufpre}(u, w) = q$. We can assume a (possibly empty) word $u'$ such that $u = u' \cdot q$, $q$ is a prefix of $w$, and (*Maximality:* ) for every suffix $s \neq \epsilon$ of $u'$, $s \cdot q$ is not a prefix of $w$.

We need to show $\mathsf{sufpre}(u' \cdot q \cdot a, w) = \mathsf{sufpre}(q \cdot a, w)$. Obviously, $\mathsf{sufpre}(q \cdot a, w)$ is a suffix of $\mathsf{sufpre}(u' \cdot q \cdot a, w)$. Assume $|\mathsf{sufpre}(u' \cdot q \cdot a, w)| > |\mathsf{sufpre}(q \cdot a, w)|$. Then, there exists a suffix $s \neq \epsilon$ of $u'$ such that $s \cdot q \cdot a$ is a prefix of $w$, contradicting the above *Maximality* condition.

We conclude by contradiction $\mathsf{sufpre}(u' \cdot q \cdot a, w) = \mathsf{sufpre}(q \cdot a, w)$. □

We now give Lemma 7.3 and then illustrate its construction by an example.

**Lemma 7.3** *Let* $M = A_1 \cdots A_n$ *be a multiword. Let* $w$ *be a nonempty word. Let* $\langle S_0, S_1, \ldots, S_n \rangle$ *be a sequence such that* $S_0 = \{\epsilon\}$ *and for every* $i \in \{1, \ldots, n\}$,

$$S_i = \lfloor \mathsf{sufpre}(S_{i-1} \cdot A_i, w) \setminus \{w\} \rfloor.$$

*Let* $m \in \{1, 2, \ldots, n\}$*. For every word* $u \in \mathsf{words}(A_1 \cdots A_m)$*,* $S_m$ *contains a suffix of* $\mathsf{sufpre}(u, w)$ *or* $u \Vdash w$*, i.e.* $w$ *is a factor of* $u$.

Proof. Proof by induction on $m$. For the induction basis, take $m = 1$ and let $u = b \in \mathsf{words}(A_1)$ such that $b \nVdash w$ (i.e. $w \neq b$). Then, $S_1$ contains a suffix of $\mathsf{sufpre}(\epsilon \cdot b, w) = \mathsf{sufpre}(b, w)$.

For the induction step, assume w.l.o.g. $S_m = \{p_1, \ldots, p_k\}$ and $A_{m+1} = \{a_1, \ldots, a_l\}$. Then, $S_{m+1} = \lfloor \{\mathsf{sufpre}(p_1 \cdot a_1, w), \ldots, \mathsf{sufpre}(p_i \cdot a_j, w), \ldots, \mathsf{sufpre}(p_k \cdot a_l, w)\} \setminus \{w\} \rfloor$. Let $u \in \mathsf{words}(A_1 \cdots A_{m+1})$. We can assume $v \in \mathsf{words}(A_1 \cdots A_m)$ and $j \in \{1, \ldots, l\}$ such that $u = v \cdot a_j$. We need to show that $S_{m+1}$ contains a suffix of $\mathsf{sufpre}(v \cdot a_j, w)$ or $v \cdot a_j \Vdash w$. Two cases can occur:

- $v \Vdash w$. Obviously, $v \cdot a_j \models w$.
- $v \nVdash w$. By the induction hypothesis, $S_m$ contains a suffix of $\mathsf{sufpre}(v, w)$. We can assume w.l.o.g. $i \in \{1, \ldots, k\}$ such that $p_i$ is a suffix of $\mathsf{sufpre}(v, w)$. We can assume word $q$ such that $\mathsf{sufpre}(v, w) = q \cdot p_i$. By Lemma 7.2, $\mathsf{sufpre}(v \cdot a_j, w) = \mathsf{sufpre}(q \cdot p_i \cdot a_j, w)$. If $\mathsf{sufpre}(p_i \cdot a_j, w) = w$, then $v \cdot a_j \Vdash w$; otherwise $S_{m+1}$ contains a suffix of $\mathsf{sufpre}(p_i \cdot a_j, w)$. Clearly, every suffix of $\mathsf{sufpre}(p_i \cdot a_j, w)$ is a suffix of $\mathsf{sufpre}(v \cdot a_j, w)$.

This concludes the proof. □

We show in Theorem 7.2 that the construction of Lemma 7.3 solves the membership problem for $\mathsf{CERTAIN}(w)$ as follows: $M \in \mathsf{CERTAIN}(w)$ if and only if the last element of $\langle S_0, S_1, \ldots, S_n \rangle$ is the empty set, with $\langle S_0, S_1, \ldots, S_n \rangle$ as in the statement of Lemma 7.3.

| | |
|---|---|
| | $S_0 = \{\epsilon\}$ |
| $A_1 = \{a\}$ | $S_1 = \{a\}$ |
| $A_2 = \{b\}$ | $S_2 = \{ab\}$ |
| $A_3 = \{d\}$ | $S_3 = \{abd\}$ |
| $A_4 = \{a\}$ | $S_4 = \{abda\}$ |
| $A_5 = \{b\}$ | $S_5 = \{abdab\}$ |
| $A_6 = \{c\}$ | $S_6 = \{abdabc\}$ |
| $A_7 = \{a\}$ | $S_7 = \{abdabca\}$ |
| $A_8 = \{a, b\}$ | $S_8 = \{a\}$ |
| $A_9 = \{b\}$ | $S_9 = \{ab\}$ |
| $A_{10} = \{d\}$ | $S_{10} = \{abd\}$ |
| $A_{11} = \{a\}$ | $S_{11} = \{abda\}$ |
| $A_{12} = \{b\}$ | $S_{12} = \{abdab\}$ |
| $A_{13} = \{c, d\}$ | $S_{13} = \{abdabc, abd\}$ |
| $A_{14} = \{a\}$ | $S_{14} = \{abdabca, abda\}$ |
| $A_{15} = \{b\}$ | $S_{15} = \{abdab\}$ |
| $A_{16} = \{c\}$ | $S_{16} = \{abdabc\}$ |
| $A_{17} = \{a\}$ | $S_{17} = \{abdabca\}$ |
| $A_{18} = \{b\}$ | $S_{18} = \{\}$ |

Figure 7.2: Illustration of the construction in Lemma 7.3.

**Example 7.4** The construction is illustrated in Figure 7.2 for the multiword

$$M = abdabca\{a, b\}bdab\{c, d\}abcab$$

and the word $w = abdabcab$. The set $S_8$, for example, is computed from $S_7 \cdot A_8 = \{abdabcaa, w\}$, in which $abdabcaa$ is replaced with its suffix $a$, and the word $w$ is removed. $\triangleleft$

A rough time complexity analysis shows that the construction of $\langle S_0, \dots, S_n\rangle$ in Lemma 7.3 is in $\mathcal{O}(|M| \cdot |\Sigma| \cdot |w|^3)$. The length of the sequence is equal to $|M| + 1$. Each $S_i$ contains at most $|w|$ prefixes of $w$, and each $A_i$ contains at most $|\Sigma|$ symbols. Each $S_i$ is of the form $\lfloor R_i \rfloor$, where $R_i$ is constructed by computing $\mathsf{sufpre}(p \cdot a, w)$ for every $p \in S_{i-1}$ and $a \in A_i$. A naive computation of $\mathsf{sufpre}(u, w)$ with $|u| \leq |w|$ takes time $\mathcal{O}(|w|^2)$, and the computation of $\lfloor R_i \rfloor$ is in $\mathcal{O}(|w|^3)$.

**Lemma 7.4** *Let $M = A_1 \cdots A_n$, $w$, and $\langle S_0, S_1, \dots, S_n\rangle$ be as in Lemma 7.3. Let $i \in \{1, \dots, n\}$. For every $p \in S_i$, there exists $u \in \mathsf{words}(A_1 \cdots A_i)$ such that $u \nVdash w$ and $\mathsf{sufpre}(u, w) = p$.*

Proof. Proof by induction on $i$. The basis of the induction, $i = 1$, is easy.

For the induction step, assume $p \in S_{i+1}$. We can assume $q \in S_i$ and $a \in A_{i+1}$ such that $w \neq p = \mathsf{sufpre}(q \cdot a, w)$. By the induction hypothesis, there exists $u \in \mathsf{words}(A_1 \cdots A_i)$ such that $u \nVdash w$ and $\mathsf{sufpre}(u, w) = q$. Then, $u \cdot a \in \mathsf{words}(A_1 \cdots A_{i+1})$. It suffices to show that $u \cdot a \nVdash w$ and $\mathsf{sufpre}(u \cdot a, w) = p$.

- Assume $u \cdot a \Vdash w$. Then necessarily, $w = q \cdot a$. But then $p = w$, a contradiction. We conclude by contradiction that $u \cdot a \nVdash w$.
- By Lemma 7.2, $\mathsf{sufpre}(u \cdot a, w) = \mathsf{sufpre}(q \cdot a, w) = p$.

This concludes the proof. $\square$

**Theorem 7.2** *Let $M = A_1 \cdots A_n$, $w$, and $\langle S_0, S_1, \ldots, S_n \rangle$ be as in Lemma 7.3. Then, $M \in \mathsf{CERTAIN}(w)$ if and only if $S_n = \{\}$.*

PROOF. *If part.* Assume $S_n = \{\}$. Lemma 7.3 implies that $M \in \mathsf{CERTAIN}(w)$.
*Only-if part.* Assume $S_n \neq \{\}$. Lemma 7.4 implies that $M \notin \mathsf{CERTAIN}(w)$. $\square$

Intuitively, the construction of $\langle S_0, S_1, \ldots, S_n \rangle$ for a word (or pattern) $w$ and a multiword $M$ can be thought of as executing the Knuth-Morris-Pratt pattern matching algorithm simultaneously on every word in $\mathsf{words}(M)$. In particular, in case $M$ is of the form $\{a_1\}\{a_2\} \cdots \{a_n\}$, i.e. every symbol is a singleton, then $\mathsf{words}(M)$ is the singleton $\{a_1 a_2 \ldots a_n\}$ and our algorithm executes in a way that is close to the Knuth-Morris-Pratt algorithm.

CHAPTER 8

# Aperiodicity of $\mathsf{CERTAIN}(w)$

It is an open conjecture that $\mathsf{CERTAIN}(w)$ is first-order definable for every word $w$. Since $\mathsf{CERTAIN}(w)$ is regular, we have that $\mathsf{CERTAIN}(w)$ is first-order definable if and only if it is aperiodic. In this chapter, we study the aperiodicity of $\mathsf{CERTAIN}(w)$ and $\mathsf{CERTAIN}_\diamond(w)$.

We first show in Section 8.1 that $\mathsf{CERTAIN}_\diamond(w)$ is aperiodic for every word $w$ if the alphabet contains at least 3 symbols. For alphabets of smaller size, the notions of multiwords and partial words coincide and $\mathsf{CERTAIN}_\diamond(w)$ is aperiodic if and only if $\mathsf{CERTAIN}(w)$ is aperiodic.

Section 8.2 introduces four families of words. Section 8.3 shows that every family contains a word that does not belong to any of the three other families.

Sections 8.5 through 8.8 will show the aperiodicity of $\mathsf{CERTAIN}(w)$ for all words $w$ in each of these four classes. The aperiodicity proofs rely on an important tool, called the Decomposition Lemma, which is proved in Section 8.4.

In Section 8.9, we study the size of those families for different alphabets and several lengths of words. We show experimentally that the four families cover a large proportion of words.

This chapter is an extended version of the following scientific publications:

**(i)** "*On First-Order Query Rewriting for Incomplete Database Histories*" in 16th International Symposium on Temporal Representation and Reasoning, 2009, IEEE Computer Society [Bruyère et al. 2009];

**(ii)** "*An Aperiodicity Problem for Multiwords*" in RAIRO – Theoretical Informatics and Applications, 46, 2012 [Bruyère et al. 2012].

## 8.1 The Case of $\mathsf{CERTAIN}_\diamond(w)$

We show that if $|\Sigma| \geq 3$, then $\mathsf{CERTAIN}_\diamond(w)$ is aperiodic for every word $w$. For alphabets of smaller size, multiwords and partial words coincide and $\mathsf{CERTAIN}(w)$ is aperiodic if and only if $\mathsf{CERTAIN}_\diamond(w)$ is aperiodic.

We begin with a lemma dealing with multiwords $M$ containing a symbol with at least three values.

**Lemma 8.1** *For every multiword $M$, if $M \in \mathsf{CERTAIN}(w)$ then, for every position $i$ in $M$ such that $|M_i| \geq 3$, we have $M_1 \cdots M_{i-1} \in \mathsf{CERTAIN}(w)$ or $M_{i+1} \cdots M_{|M|} \in \mathsf{CERTAIN}(w)$.*

Proof. Let $M$ be a multiword and $w$ a word such that $M \in \mathsf{CERTAIN}(w)$. Let $i$ be a position in $M$ such that $|M_i| \geq 3$. Let $a, b, c$ be three distinct symbols in $M_i$. Assume $M \in \mathsf{CERTAIN}(w)$, $M_1 \cdots M_{i-1} \notin \mathsf{CERTAIN}(w)$ and $M_{i+1} \cdots M_n \notin \mathsf{CERTAIN}(w)$.

Let $m \in \mathsf{words}(M_1 \cdots M_{i-1})$ and $m' \in \mathsf{words}(M_{i+1} \cdots M_n)$ such that neither $m$ nor $m'$ contains $w$ as a factor. By hypothesis, $w$ is a factor of $mam', mbm', mcm'$. Therefore, w.l.o.g., there exist words $u, v, x, y$ such that $w = uavbxcy$ and:

- $u$ is a suffix of $m$ and $vbxcy$ is a prefix of $m'$;
- $uav$ is a suffix of $m$ and $xcy$ is a prefix of $m'$; and
- $uavbx$ is a suffix of $m$ and $y$ is a prefix of $m'$.

Graphically,

| $\leftarrow m \rightarrow$ | | $\leftarrow m' \rightarrow$ |
|---:|:---:|:---|
| $\cdots u$ | $a$ | $vbxcy \cdots$ |
| $\cdots uav$ | $b$ | $xcy \cdots$ |
| $\cdots uavbx$ | $c$ | $y \cdots$ |

Let $k = |u| + 1 + |x| + 1$. If we start reading the first line in the array above, and switch to the second one after $m$, we observe that the $k$-th symbol of $w$ must be $c$. Let $j = |y| + 1 + |v| + 1$. Then, if we start reading the third line from the end of $w$, and switch to the second one after reading $m'$, we note that the $j$-th last symbol of $w$ must be $a$. Since $k + j = |w| + 1$, the $k$-th symbol equals the $j$-th last symbol, hence $a = c$, a contradiction. This concludes the proof. $\square$

Since $\Sigma$ contains at least 3 symbols, Lemma 8.1 can be applied for every position in a partial word that is not a singleton.

**Theorem 8.1** *If $|\Sigma| \geq 3$, then $\mathsf{CERTAIN}_\diamond(w)$ is aperiodic for each $w \in \Sigma^+$.*

Proof. Let $\Sigma$ be an alphabet of at least 3 symbols. Let $w$ be a word. Let $M$ be a multiword that is a partial word. By repeated applications of Lemma 8.1 on $M$, it is easy to see that $M \in \mathsf{CERTAIN}_\diamond(w)$ if and only if $\langle\{w_1\}, \{w_2\}, \ldots, \{w_{|w|}\}\rangle$ is a factor of $M$. It follows that $M \in \mathsf{CERTAIN}_\diamond(w)$ if and only if $\mathsf{struct}(M)$ satisfies the following first-order formula.

$$\exists i_1 \exists i_2 \ldots \exists i_{|w|} \Big( S(i_1, i_2) \wedge \cdots \wedge S(i_{|w|-1}, i_{|w|}) \wedge P_{\{w_1\}}(i_1) \wedge P_{\{w_2\}}(i_2) \wedge \cdots \wedge P_{\{w_{|w|}\}}(i_{|w|}) \Big).$$

We have that $\mathsf{CERTAIN}_{\diamond}(w)$ is first-order definable. The aperiodicity follows by Theorem 7.1. □

## 8.2 Four Families of Words

We show the aperiodicity of $\mathsf{CERTAIN}(w)$ for every word $w$ in four large families of words. This section introduces those families. We show in Section 8.3 that those families are incomparable under set inclusion. Sections 8.5 through 8.8 show aperiodicity of $\mathsf{CERTAIN}(w)$ for each family.

### Family $\mathcal{F}_{\text{rep.3}}$ of Repeated ($\geq 3$) Words

The first family of words contains every word $w$ such that $w$ is at least three repetitions of some word $u$.

**Definition 8.1** We denote by $\mathcal{F}_{\text{rep.3}}$ the family of words of the form $pu^kq$ where

1. $u$ is a nonempty word,
2. $k \geq 3$,
3. $p$ is a proper suffix of $u$, and $q$ is a proper prefix of $u$.

◁

A word $w \in \Sigma^+$ is *primitive* if $w = v^k$ implies $k = 1$. Every word in $\mathcal{F}_{\text{rep.3}}$ can be viewed as a power ($\geq 3$) of a primitive word. This is Lemma 8.2

**Lemma 8.2** *Every word $w \in \mathcal{F}_{\text{rep.3}}$ is of the form $w = v^h v'$ where*

1. *$v$ is a nonempty primitive word,*
2. *$h \geq 3$, and*
3. *$v'$ is a proper prefix of $v$.*

Proof. We make use of the following sublemmas.

**Sublemma 8.1** *[Lothaire 1997] Every word is a power ($\geq 1$) of a primitive word.*

**Sublemma 8.2** *Every word $w \in \mathcal{F}_{\text{rep.3}}$ is of the form $w = v^h v'$ where $v$ is a word, $v'$ is a proper prefix of $v$ and $h \geq 3$.*

PROOF. Let $w = pu^kq$ following Definition 8.1. Let $p'$ be a word such that $u = p'p$. We have $w = p(p'p)^kq = (pp')^kpq$. Let $v = pp'$. We distinguish two cases.

- Case $|q| < |p'|$. Since $q$ is a prefix of $p'$, $pq$ is a prefix of $pp' = v$. It follows $w = v^kv'$ with $v' = pq$.
- Case $|q| > |p'|$. It must be the case that $pq = pp'v'$ for some proper prefix $v'$ of $p$. It follows $w = v^{k+1}v'$.

$\dashv$

Let $w \in \mathcal{F}_{\text{rep.3}}$. By Sublemma 8.2, $w = v^hv'$ where $v$ is a word, $v'$ is a proper prefix of $v$ and $h \geq 3$. By Sublemma 8.1, $v = u^j$ for some $u$ primitive, $j \geq 1$. If $j = 1$, the desired result follows. Assume $j > 1$. We distinguish two cases.

- Case $|v'| < |u|$. We have $w = u^{jh}v'$ with $v'$ a proper prefix of $u$ and $jh \geq h \geq 3$.
- Case $|v'| \geq |u|$. It must be the case that $v' = u^nu'$ for some $n \geq 1$ and $u'$ a proper prefix of $u$. Then, $w = u^{jh}u^nu' = u^{jh+n}u'$ where $jh + n \geq h \geq 3$.

This concludes the proof. $\square$

For example, the word $b(abab)^3a$ is of the form $pu^kq$ according to Definition 8.1 and is equal to $(ba)^7\epsilon$.

## Family $\mathcal{F}_{\text{p.unb.}}$ of Powers of Unbordered Words

The second family is composed of powers of unbordered words.

**Definition 8.2** A nonempty word $u \in \Sigma^+$ is a *border* of a word $w \in \Sigma^*$ if $w = uv = v'u$ for some nonempty words $v$ and $v'$. If a word has no border, it is *unbordered*, otherwise it is *bordered*. $\lhd$

For example, the word $ababa$ has two borders: $a$ and $aba$. The word $aaab$ is unbordered.

**Definition 8.3** We denote by $\mathcal{F}_{\text{p.unb.}}$ the family of words $w$ such that, for some unbordered word $v$ and integer $h \geq 1$, we have $w = v^h$. $\lhd$

The family $\mathcal{F}_{\text{p.unb.}}$ includes both bordered and unbordered words. For example, $\mathcal{F}_{\text{p.unb.}}$ contains both the unbordered word $ab$ and the bordered word $(ab)^2$.

## Family $\mathcal{F}_{\text{anch.}}$ of Anchored Words

The third family is composed of "anchored" words.

**Definition 8.4** A word $w$ is anchored if $w = savat$ for some $a \in \Sigma$, and $v, s, t \in \Sigma^*$ such that

1. $v$ does not contain $a$;

$$\begin{array}{cccc|ccccc|ccccccc}
 & \underline{a} & \underline{a} & b & & & \underline{a} & a & b & & & & \underline{a} & b & a & b \\
a & a & b & & a & a & b & & & a & b & a & b & & &
\end{array}$$

Figure 8.1: Rims for *aab* and *abab*.

2. $ava$ occurs only once in $w$: if $w = s_1avat_1 = s_2avat_2$ then $s_1 = s_2$;

3. if $s \neq \epsilon$, no nonempty prefix of $w$ is a suffix of $ava$; and

4. if $t \neq \epsilon$, no nonempty suffix of $w$ is a prefix of $ava$.

Such a word $w$ is called *anchored* with $ava$ as an *anchor*. We denote by $\mathcal{F}_{\text{anch.}}$ the family of anchored words. $\lhd$

For example, $w = b^2abab^2$ is an anchored word with $aba$ as an anchor. Word $aba$ is also anchored. Intuitively, the anchor $ava$ of an anchored word $savat$ restrict the ways the word can overlap with itself. The role of the anchor will be highlighted in the proof of Theorem 8.4.

## Family $\mathcal{F}_{\text{unr.}}$ of Unrimmed Words

For the fourth family of words, we define a variant of the notion of border called a *rim*. Intuitively, a rim of a word $w$ is a nonempty prefix of $w$ such that $w$ has a suffix of the same length and there is exactly one mismatch between the prefix and the suffix.

**Definition 8.5** A nonempty word $u \in \Sigma^+$ is a *rim* of a word $w \in \Sigma^*$ if $w = uv = v'u'$ for some nonempty words $v, v'$ and $u'$ such that $|u| = |u'|$ and $u$ and $u'$ agree on all positions except one. A word is *unrimmed* if it has no rim. We denote by $\mathcal{F}_{\text{unr.}}$ the family of unrimmed words. $\lhd$

For example, word $aab$ has two rims ($a$ and $aa$) and word $abab$ has one rim ($a$). Figure 8.1 shows these rims, underlined for clarity. The family $\mathcal{F}_{\text{unr.}}$ contains, for example, the words $aabaa$ and $abca$. Clearly, every unrimmed word agrees on its first and last position, and hence is bordered.

Remember that a palindrome is a word that cannot be distinguished when read from left to right and from right to left. The following lemma establishes that every palindrome belongs to $\mathcal{F}_{\text{unr.}}$. Note that the converse is not true. For instance, $aababbaa$ is unrimmed but is not a palindrome.

**Lemma 8.3** *Every palindrome is unrimmed.*

Proof. Let $w$ be a palindrome. We show by induction on the length of $w$ that $w$ has no rim.

- Base case $w = a$ or $w = aa$. Obvious.

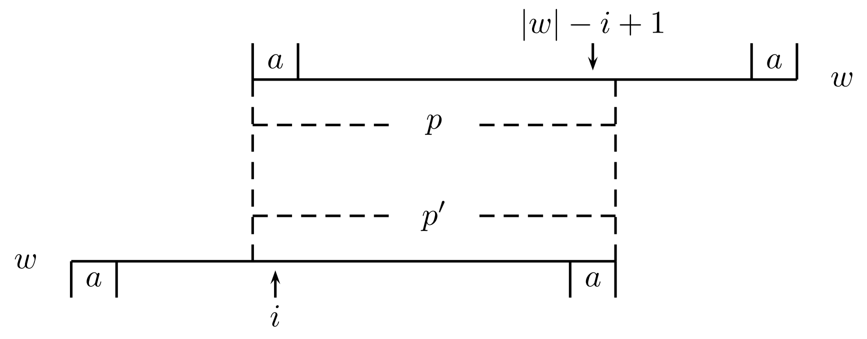

Figure 8.2: Situation in the proof of Lemma 8.3.

- Assume $w = ava$ where $v$ is a palindrome. By contradiction, assume $w$ has a rim. The alignment is depicted in Figure 8.2. In this figure, $p$ is a rim for $w$, meaning there exists exactly one mismatch between the rim $p$ and its corresponding suffix $p'$. Let $i = |w| - |p'|$ be a position in $w$, in front of the first $a$. As $w$ is a palindrome, we have $w_i = w_{|w|-i-1}$. Two cases can occur:
  - Suppose $w_i = a$. By the induction hypothesis, we know $v$ has no rim and it must be the case that there is no mismatch between $p$ and $p'$, a contradiction.
  - Suppose $w_i = b$. Then, $p = aqb$ and $p' = bq'a$ for some suitable words $q$ and $q'$. But in that case, the number of mismatches between $p$ and $p'$ is at least 2, a contradiction.

We conclude by contradiction that $w$ has no rim. □

## 8.3 Comparisons of the Four Families

The following lemma states that anchored words are primitive words and that no anchored word belongs to $\mathcal{F}_{\text{rep.3}}$.

**Lemma 8.4**

*(1) Every anchored word is primitive.*

*(2) $\mathcal{F}_{rep.3} \cap \mathcal{F}_{anch.} = \emptyset$.*

Proof. (First item) Let $w$ be a word such that $w$ is anchored. Assume by contradiction that $w$ is not primitive. Then $w = u^h$ for some $h > 1$. But in this case, the anchor $ava$ cannot appear in $u$ by condition (2) of Definition 8.4, so there exists $i$, $0 \leq i \leq |v|$, such that $av_1 \cdots v_i$ is a suffix of $u$ and $v_{i+1} \cdots v_{|v|}a$, a prefix of $u$: this contradicts conditions (3) and (4) of Definition 8.4.

(Second item) Anchored words do not belong to $\mathcal{F}_{\text{rep.3}}$: if $w = u^h u'$ with $u$ primitive, $h \geq 3$ and $u'$ a proper prefix of $u$ (see Lemma 8.2), then an anchor $ava$ cannot appear in $u$ nor $u^2$ by condition (2) of Definition 8.4, and can neither appear in $uu'$ (it would appear in $u^2$). Assume $ava$ appears in $u^3$ and not in $u^2$. Then, $|u| < |ava|$ and it must be the case that $v$ contains $a$, which contradicts condition (1). $\square$

The four families of words are incomparable under set inclusion: every family contains a word that belongs to no other family. For instance, *abaabaabaa* is only in $\mathcal{F}_{\text{rep.3}}$, *aaab* is only in $\mathcal{F}_{\text{p.unb.}}$, *aaabba* only in $\mathcal{F}_{\text{anch.}}$, and *abca* only in $\mathcal{F}_{\text{unr.}}$. Finally, word *aaba* does not belong to any of those families.

Starting from our four languages, we can construct $2^4 = 16$ languages by means of intersect and complement. Table 8.1 lists these 16 languages. For example, the second line corresponds to $\overline{\mathcal{F}_{\text{rep.3}}} \cap \overline{\mathcal{F}_{\text{p.unb.}}} \cap \overline{\mathcal{F}_{\text{anch.}}} \cap \mathcal{F}_{\text{unr.}}$. The following proposition shows that $\overline{\mathcal{F}_{\text{rep.3}}} \cap \mathcal{F}_{\text{p.unb.}} \cap \overline{\mathcal{F}_{\text{anch.}}} \cap \mathcal{F}_{\text{unr.}}$ only contains words of length equal to 1 (line 6 in Table 8.1).

**Proposition 8.1** *Let $w$ be a word. If $w \in \overline{\mathcal{F}_{\textit{rep.3}}} \cap \mathcal{F}_{\textit{p.unb.}} \cap \overline{\mathcal{F}_{\textit{anch.}}} \cap \mathcal{F}_{\textit{unr.}}$, then $w = a$ for some $a \in \Sigma$.*

Proof. Let $w \in \mathcal{F}_{\text{p.unb.}}$, this is, $w = v^h$ where $v$ is unbordered and $h \geq 1$. We distinguish two cases:

- Case $|v| \geq 2$. As $v$ is unbordered, we can assume w.l.o.g. that $v$ starts with $a \in \Sigma$ and ends with $b \in \Sigma$. This implies that $v$ and $w$ have a rim, a contradiction.
- Case $|v| = 1$. If $h \geq 3$, we have $w \in \mathcal{F}_{\text{rep.3}}$, a contradiction. If $h = 2$, we have $w = aa$ for some $a \in \Sigma$ and $w$ is anchored, a contradiction. If $h = 1$, the desired result follows.

We conclude that $w = a$ for some $a \in \Sigma$. $\square$

## 8.4 Decomposition Lemma

Our main aperiodicity results are stated and proven in the following sections. In each case, aperiodicity is established by using Lemma 7.1, *ad absurdum*, in combination with a decomposition lemma. This section is devoted to this lemma which is our main technical tool.

**Lemma 8.5 (Decomposition Lemma)** *Let $w \in \Sigma^+$ be a word, and $k \geq 1$. Let $P, U, Q \in \widehat{\Sigma}^*$ be multiwords such that $P\ U^k\ Q \models_{\mathsf{certain}} w$. Let $p \in \mathsf{words}(P)$, $q \in \mathsf{words}(Q)$ and $u \in \mathsf{words}(U^{k+1})$. If $m = puq$ does not contain $w$ as a factor, then for every position $\pi$ in $m$ such that $|P| \leq \pi \leq |PU^k|$, there exist $x, y \in \Sigma^+$ such that*

1. *$w = xy$;*
2. *$x$ is a nonempty suffix of $m_1 \cdots m_\pi$; and*
3. *$y$ is a nonempty prefix of $m_{\pi+|U|+1} \cdots m_{|m|}$.*

| $\mathcal{F}_{\text{rep.3}}$ | $\mathcal{F}_{\text{p.unb.}}$ | $\mathcal{F}_{\text{anch.}}$ | $\mathcal{F}_{\text{unr.}}$ | Word in the language |
|---|---|---|---|---|
| | | | | $aaba$ |
| | | | $\in$ | $aabaa$ |
| | | $\in$ | | $aaabba$ |
| | | $\in$ | $\in$ | $aababaa$ |
| | $\in$ | | | $aaab$ |
| | $\in$ | | $\in$ | $a$ |
| | $\in$ | $\in$ | | $ababb$ |
| | $\in$ | $\in$ | $\in$ | $aa$ |
| $\in$ | | | | $(aba)^3a$ |
| $\in$ | | | $\in$ | $(ab)^3a$ |
| $\in$ | | $\in$ | | empty by Lemma 8.4 |
| $\in$ | | $\in$ | $\in$ | empty by Lemma 8.4 |
| $\in$ | $\in$ | | | $(ab)^3$ |
| $\in$ | $\in$ | | $\in$ | $a^3$ |
| $\in$ | $\in$ | $\in$ | | empty by Lemma 8.4 |
| $\in$ | $\in$ | $\in$ | $\in$ | empty by Lemma 8.4 |

Table 8.1: Summary of the membership to different families of words.

In other words, this lemma states that, for every position $\pi$ of $m$ (under the hypotheses), there exist a prefix $x$ of $w$ ending at position $\pi$ and a suffix $y$ of $w$ beginning at position $\pi + |U| + 1$, such that $xy = w$. This situation is depicted in Figure 8.3.

The pair $(x, y)$ mentioned in Lemma 8.5 is called an *$\ell$-decomposition of $w$ at position* $\pi$ (or simply an $\ell$-decomposition at position $\pi$, if $w$ is clear from the context), and also an *$r$-decomposition of $w$ at position* $\pi + |U| + 1$.[1] A position $\pi$ is *left-maximal* if $m_{\pi+1} \neq m_{\pi+1+|U|}$. Any $\ell$-decomposition $(x, y)$ at a left-maximal position is called *maximal.* A position $\pi$ is *right-maximal* if $m_{\pi-1} \neq m_{\pi-1-|U|}$. Any $r$-decomposition $(x, y)$ at a right-maximal position is called *maximal.* Finally, a *witness* is a maximal $\ell$-decomposition of $w$ at a left-maximal position, or a maximal $r$-decomposition of $w$ at a right-maximal position. The rationale for calling decompositions maximal comes from the following obvious observations:

- If $(x, y)$ is an $\ell$-decomposition of $w$ at a left-maximal position $\pi$, then $xm_{\pi+1}$ is not a prefix of $w$. Intuitively, $x$ cannot be extended to the right.
- If $(x, y)$ is an $r$-decomposition of $w$ at a right-maximal position $\pi$, then $m_{\pi-1}y$ is not a suffix of $w$. Intuitively, $y$ cannot be extended to the left.

Consider for instance the case $w = abab$. Figure 8.4 shows two $\ell$-decompositions $(a, bab)$ and $(aba, b)$ at a position $\pi$ of a word $m$. Both $\ell$-decompositions are maximal, since none of them can be extended to position $\pi + 1$. Hence $\pi$ is a left-maximal position, with witnesses $(a, bab)$ and $(aba, b)$.

[1]Notice that in the $\ell$-decomposition, $x$ ends at position $\pi$, and in the $r$-decomposition, $y$ begins at position $\pi + |U| + 1$ (See Figure 8.3).

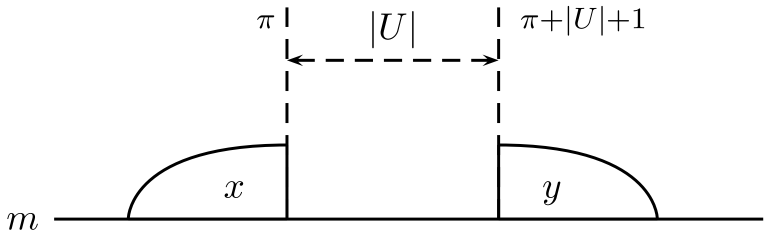

Figure 8.3: $\ell$-decomposition $(x, y)$ of $w$ at position $\pi$.

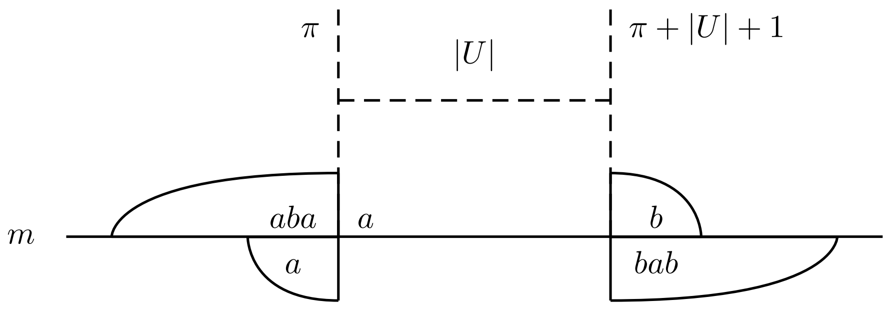

Figure 8.4: Two maximal $\ell$-decompositions.

Proof.[Proof of Lemma 8.5] Let $\pi$ be a position in $m$ such that $|P| \leq \pi \leq |PU^k|$. We can assume $m = pv_1uv_2q$ with $|pv_1| = \pi$ and $|u| = |U|$. Let $m' = pv_1v_2q$. From $PU^kQ \models_{\mathsf{certain}} w$ and $m' \in \mathsf{words}(PU^kQ)$, we have $m' \Vdash w$. The situation is:

$$m = \overbrace{pv_1uv_2q}^{\not\Vdash w} \qquad m' = \underbrace{\overbrace{pv_1}^{\not\Vdash w}\ \overbrace{v_2q}^{\not\Vdash w}}_{\Vdash w}$$

But $m \not\Vdash w$ implies $pv_1 \not\Vdash w$ and $v_2q \not\Vdash w$, so $pv_1$ ends with some nonempty prefix $x$ of $w$, and $v_2q$ starts with some nonempty suffix $y$ of $w$, and $w = xy$. □

The following lemma shows that every $\ell$-decomposition can be extended to the right, until a maximal $\ell$-decomposition is reached.

**Lemma 8.6** *Let $w, k, P, U, Q, m$ and $\pi$ be defined as in Lemma 8.5. Let $(x, y)$ be an $\ell$-decomposition of $w$ at position $\pi$. There exists a left-maximal position $\pi + j$ such that $0 \leq j < |y|$ and with a witness $(x', y')$ where $x' = xm_{\pi+1} \cdots m_{\pi+j}$.*

*Symmetrically, if $(x, y)$ is an r-decomposition at position $\pi$, then there exists a right-maximal position $\pi - j$ such that $0 \leq j < |x|$ and with a witness $(x', y')$ where $y' = m_{\pi-j} \cdots m_{\pi-1}y$.*

Proof. Suppose for contradiction that for all $j$ satisfying $0 \leq j < |y|$, the position $\pi + j$ is not left-maximal. We show that under these conditions, $w$ is a factor of $m$, which is impossible. Let $a$ be the first symbol of $y$, and $y'$ be such that $y = ay'$. As $\pi$ is not left-maximal, $x$ can be extended to $x' = xa$, and $(x', y')$ is an $\ell$-decomposition of $w$ at position $\pi + 1$.

We can repeat this step from $\pi + 1$ to $\pi + 2$, and so on, $|y|$ times. Thus, $x$ can be extended symbol by symbol, until $w$ appears as factor. □

**Remark 8.1** *In the remainder, we will show the aperiodicity of $\mathsf{CERTAIN}(w)$ ad absurdum. We will apply Lemma 8.5 at several positions $\pi$* "far enough from extremities" *in some possible word $m$, without explicitly checking the condition $|P| \leq \pi \leq |PU^k|$. In fact, all our proofs are local, in that they work on a* region *of the word $m$, which length only depends on $|w|$ and $|U|$. Let us check that we can always find such a region, where Lemma 8.5 can be applied.*

*We can define such a region as an interval between a leftmost position $\pi_1$ and a rightmost position $\pi_2$. As mentioned above, the width of the region only depends on $|w|$ and $|U|$, that is $\pi_2 = \pi_1 + i|w| + j|U|$ for some $i, j \geq 0$ depending on the proof we consider. For each proof, $i$ and $j$ are fixed, and for every $w \in \Sigma^+$, we have to find $k$ such that for all $P, U, Q$, there is a position $\pi_1$ for which $|P| \leq \pi_1$ and $\pi_1 + i \cdot |w| + j \cdot |U| \leq |PU^k|$. This is equivalent to $|P| \leq \pi_1 \leq |P| + (k - j) \cdot |U| - i \cdot |w|$. As $|U| \geq 1$ (the case $|U| = 0$ being trivial), it is sufficient that $|P| \leq \pi_1 \leq |P| + (k - j) - i \cdot |w|$. Lemma 8.5 applies if $k \geq j + i \cdot |w|$.*

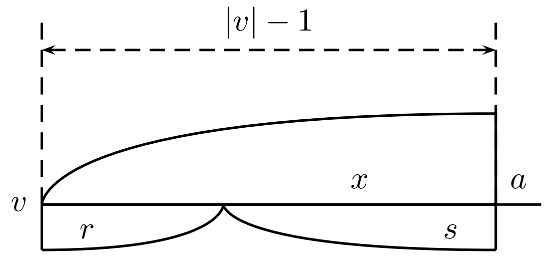

Figure 8.5: Two $v$-factorizations.

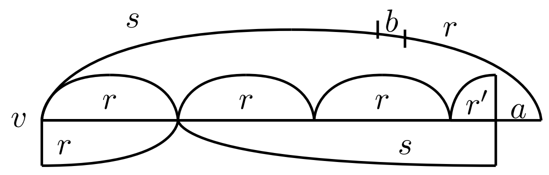

Figure 8.6: $x$ has period $r$.

## 8.5 Family $\mathcal{F}_{\text{rep.3}}$ of Repeated ($\geq 3$) Words

We show in Theorem 8.2 that $\mathsf{CERTAIN}(w)$ is aperiodic for every word $w$ in $\mathcal{F}_{\text{rep.3}}$ (see Definition 8.1). The proof is quite long and technical. We first introduce some terminology [Lothaire 1997] and some helpful lemmas.

A word $w$ is a *conjugate* of a word $w'$ if $w = uv$ and $w' = vu$ for some nonempty words $u, v$. It is known that all conjugates of a primitive word are primitive. We say that a word $x$ *has period* $r$ if $r \neq \epsilon$ and $x = r^i r'$ with $i \geq 1$ and $r'$ is a proper prefix of $r$. If $v$ is a primitive word, a *$v$-factorization* of a word $x$ is a factorization of the form $x = r \cdot v^i \cdot s$ where $i \geq 0$, $r$ is a proper suffix of $v$ and $s$ is a proper prefix of $v$. A word $x$ can have several $v$-factorizations. The following lemma shows that the $v$-factorization is unique when $x$ is large enough.

**Lemma 8.7** *Let $v$ be a primitive word. If $x$ has a $v$-factorization, and $|x| \geq |v| - 1$, then $x$ has only one $v$-factorization.*

Proof. We first consider the case where $|x| = |v| - 1$. Assume for contradiction that $x$ has two distinct $v$-factorizations. We can assume that these two $v$-factorizations are $rs$ and $\epsilon x$ (where $\epsilon$ is the empty word). This can be assumed w.l.o.g. by considering the suitable conjugate of $v$.

Hence the situation looks like in Figure 8.5, with $r$ a proper nonempty suffix of $v$, $s$ a proper prefix of $v$, and $xa = v$ where $a$ is a symbol. In particular, $s$ is prefix of $x$. Let $b$ be the symbol such that $sb$ is a proper prefix of $v$. We will show in the following that

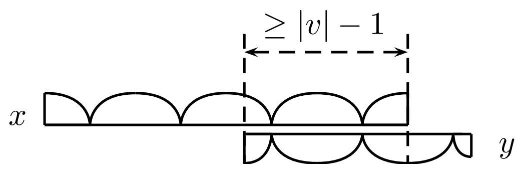

Figure 8.7: Synchronization of words $x$ and $y$.

$x$ has period $r$ and also period $sb$. Then, as $|x| = |r| + |sb| - 1$, we can apply Fine and Wilf's theorem [Lothaire 1997; Fine and Wilf 1965], to obtain that $r$ and $sb$ are powers of the same word. Hence $v = (sb)r$ is not primitive, which is a contradiction.

Let us prove that $x$ has period $r$. Figure 8.6 illustrates the situation. As $x = rs$, it is sufficient to prove that either $s$ is a proper prefix of $r$, or $s$ has period $r$. If $|s| < |r|$, then because of the two $v$-factorizations, $s$ is a proper prefix of $r$. If $|s| \geq |r|$, $r$ is now a prefix of $s$. Let $r^j$ be the prefix of $s$, with $j \geq 1$ being maximal. The word $r^{j+1}$ is a prefix of $rs$. But $s$ is also a prefix of $rs$ (consider the two $v$-factorizations). Hence, either $s$ has period $r$, or $r^{j+1}$ is a prefix of $s$. The latter is impossible by definition of $j$, so $s$ has period $r$.

Now we prove that $sb$ is a period of $x$. The proof follows the same line. As $v = (sb)r$, we only have to show that either $r$ is a proper prefix of $sb$, or $r$ has period $sb$. Then we just have to consider two cases $|r| < |sb|$ and $|r| \geq |sb|$ as we did before.

We now consider the case where $|x| \geq |v| + 1$. Note that if $x$ has two distinct $v$-factorizations, then $x$ has a factor $x'$ of length $|x| - 1$ with two distinct $v$-factorizations. This is impossible according to the preceding arguments. This concludes the proof. □

Let $v$ be a primitive word. If two words $x$ and $y$ have a $v$-factorization, and have a common factor of length $|v| - 1$, then the preceding lemma implies that their $v$-factorizations are identical on this common factor, as depicted in Figure 8.7. In this case, we say that $x$ and $y$ *are synchronized* (according to $v$).

We now show that if $w$ is in $\mathcal{F}_{\text{rep.3}}$, then CERTAIN($w$) is aperiodic. To improve readability, some parts of the proof are stated as sublemmas.

**Theorem 8.2** *For every word $w \in \mathcal{F}_{\text{rep.3}}$, CERTAIN($w$) is aperiodic.*

Proof. Let $w \in \mathcal{F}_{\text{rep.3}}$. According to Lemma 8.2, $w$ is of the form $w = v^h v'$ where $v$ is primitive, $h \geq 3$, and $v'$ is a proper prefix of $v$. We prove Theorem 8.2 by contradiction, using Lemma 7.1.

Let $k$ be sufficiently large (see Remark 8.1), and let $P, U, Q \in \widehat{\Sigma}^*$ be multiwords such that $PU^kQ \models_{\mathsf{certain}} w$. Let $p \in \mathsf{words}(P)$, $q \in \mathsf{words}(Q)$ and $u \in \mathsf{words}(U^{k+1})$. Assume for contradiction that $m = puq$ does not contain $w$ as a factor. Therefore $|U| > 0$ and Lemma 8.5 can be applied. We start the proof with two sublemmas based on these hypotheses. The first sublemma can be paraphrased as follows. Consider a left-maximal position $\pi$ with witness $(x, y)$. Then, consider any $\ell$-decomposition $(x', y')$ at the next position $\pi + 1$. The sublemma implies that $x$ and $x'$ cannot overlap much; in particular, the length of each common factor must be less than $|v| - 1$.

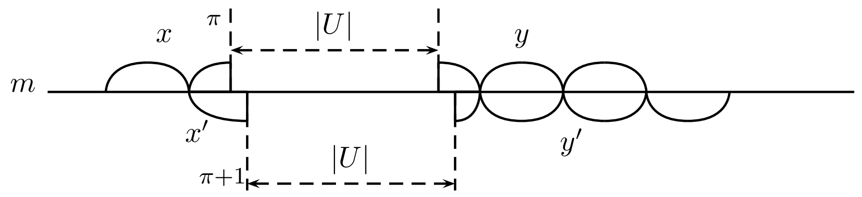

Figure 8.8: Decompositions in the proof of Sublemma 8.3, case $|x| < |v|$.

**Sublemma 8.3** *Let $\pi$ be a left-maximal position with witness $(x, y)$. Let $(x', y')$ be an $\ell$-decomposition at position $\pi + 1$. Then for all common factors $f$ of $x$ and $x'$ (resp. of $y$ and $y'$), $|f| < |v| - 1$.*

Proof. The words $x$ and $x'$ are proper prefixes of $w$, while $y$ and $y'$ are proper suffixes of $w$, so these four words have a $v$-factorization. Assume that $x$ and $x'$ share a common factor $f$ with $|f| \geq |v| - 1$. Then, by Lemma 8.7, they are synchronized. Hence $x$ can be extended to a larger prefix of $w$ (using $x'$), which contradicts the premise that $\pi$ is left-maximal. Assume now, that $y$ and $y'$ have a common factor $f$ with $|f| \geq |v| - 1$. By Lemma 8.7, they are synchronized, as illustrated in Figure 8.8. The shift of $|U|$ is the same in both decompositions, so $x$ can be extended to a longer prefix of $w$ by one symbol, which is impossible. ⊣

The second sublemma shows that a maximal $\ell$-decomposition $(x, y)$ is such that $x$ is large compared to $y$. In particular, the length of $x$ is always more than twice the length of $y$. Symmetrically for a maximal $r$-decomposition.

**Sublemma 8.4** *If $\pi$ is a left-maximal position with witness $(x, y)$, then $|x| > |w| - |v|$ and $|y| < |v|$. Symmetrically, if $\pi$ is right-maximal with witness $(x, y)$, then $|x| < |v|$ and $|y| > |w| - |v|$.*

Proof. We only prove the first part of the sublemma (the proof of the second part is symmetrical). Suppose for contradiction that $|x| \leq |w| - |v|$. We distinguish two cases: $|v| \leq |x|$ and $|x| < |v|$.

- Case $|v| \leq |x| \leq |w| - |v|$. We have $|y| \geq |v|$, because $w = xy$. Let $(x', y')$ be an $\ell$-decomposition at position $\pi + 1$. As $w = x'y'$ and $|w| \geq 3|v|$, $x, x'$ or $y, y'$ have a common factor $f$ such that $|f| \geq |v| - 1$. This is impossible according to Sublemma 8.3.

- Case $|x| < |v|$. Now we have $|y| > |w| - |v|$. As $w = v^h v'$ with $h \geq 3$, we get $|y| > 2|v|$. Let us consider an $\ell$-decomposition $(x', y')$ at position $\pi + 1$. Sublemma 8.3 tells us that $|y'| < |v|$, and thus $|x'| > |w| - |v|$. According to Lemma 8.6 applied to $x'$ at position $\pi + 1$, there exists a left-maximal position $\pi + j$ such that $1 \leq j \leq |y'|$ with witness $(x'', y'')$, $x'$ being a prefix of $x''$. Now consider an $\ell$-decomposition $(\widehat{x}, \widehat{y})$ at position $\pi + j + 1$. The situation is illustrated in Figure 8.9. Recall that $|x'| > |w| - |v|$ and $|y| > |w| - |v|$. As $|x''| \geq |x'|$, we have $|x''| > 2|v|$. Applying

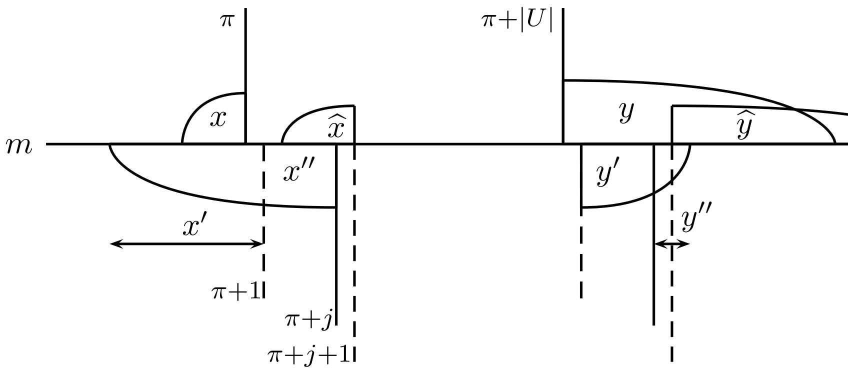

Figure 8.9: Decompositions in the proof of Sublemma 8.4.

Sublemma 8.3 at position $\pi + j$, we get $|\widehat{x}| < |v|$ and thus $|\widehat{y}| > |w| - |v|$. We also have $j + 1 \leq |y'| + 1 \leq |v|$ by Lemma 8.6. So $\widehat{y}$ and $y$ have a common factor $f$ with $|f| > |w| - 2|v| > |v|$. By Lemma 8.7, they are synchronized.

Again, two cases can occur:

– $y$ and $\widehat{y}$ end at the same position. In this case, $x$ and $\widehat{x}$ have $x$ as common prefix. Hence, $\widehat{x}$ extends $x$ to the right, yielding a larger prefix of $w$. This contradicts the premise that $\pi$ is left-maximal.

– $y$ and $\widehat{y}$ do not end at the same position. As $|\widehat{x}| < |v|$, we know that $\widehat{y}$ ends after $y$. Moreover, $y$ and $\widehat{y}$ are synchronized, so $\widehat{y}$ is obtained from $y$ by a shift of $|v|^i$ positions in $m$, for some $i \geq 1$. Let $\widetilde{v}$ be the conjugate of $v$ ending with $v'$ (recall that $w = v^h v'$). The word $y\widetilde{v}$ appears as factor of $m$. However, $|x| < |v|$, so $|y\widetilde{v}| > |w|$, and thus $w$ is a factor of $m$. This is impossible.

This concluses the proof of Sublemma 8.4. $\dashv$

We use the preceding lemmas and sublemmas to complete the proof of Theorem 8.2. By Lemma 8.6, we can assume a left-maximal position $\pi$ with witness $(x, y)$. By Sublemma 8.4, we have $|x| > |w| - |v|$ and $|y| < |v|$. Let $(x', y')$ be an $\ell$-decomposition at position $\pi + 1$. According to Sublemma 8.3, $|x'| < |v|$. Using Lemma 8.6, we can extend $x'$ until a left-maximal position $\pi'$ with witness $(x'', y'')$, where $x'$ is a prefix of $x''$. We know by Sublemma 8.4 that $|x''| > |w| - |v|$ and $|y''| < |v|$.

In the sequel, we consider the positions $\pi$, $\pi - |v|$, $\alpha$, $\beta$ and $\gamma$, as illustrated in Figure 8.10:

- $\alpha$ is the position where $x'$ starts, i.e. $x' = m_\alpha \cdots m_{\pi+1}$;
- $\beta = \alpha + |v|$;
- $\gamma = \pi - |v| + |y| + 1$ is the position of $m$ corresponding to the $(|w|-|v| + 1)$-th position in $x$.

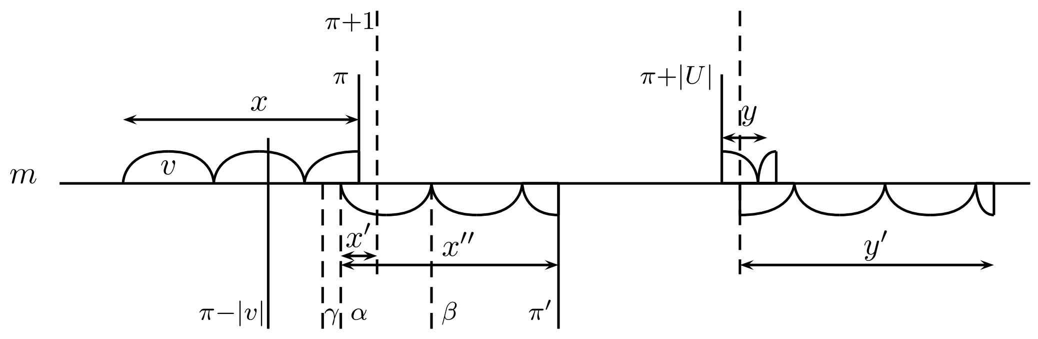

Figure 8.10: Decompositions in the proof of Theorem 8.2, represented in terms of $v$.

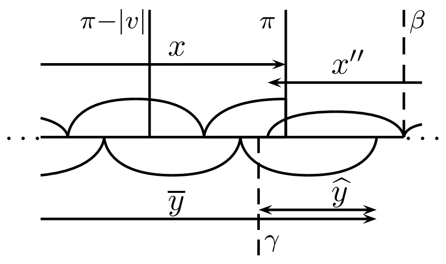

Figure 8.11: Case $|\widehat{y}| < \beta - \gamma$.

As $1 \leq |y| < |v|$, we know that:

$$\pi - |v| + 2 \leq \gamma < \pi + 1.$$

From the definition of $\alpha$ and the fact that $|x'| < |v|$, we have:

$$\pi - |v| + 2 < \alpha \leq \pi + 1.$$

We distinguish two cases, and show that both of them lead to a contradiction. Let $(\widehat{x}, \widehat{y})$ be an $r$-decomposition at position $\gamma$.

- Case $|\widehat{y}| < \beta - \gamma$ (i.e. $\widehat{y}$ ends before position $\beta - 1$).

  Then $\gamma$ is not a right-maximal position. Indeed, $|\widehat{y}| < \beta-\gamma < 2|v|$ because $\beta = \alpha+|v|$ and $\alpha - \gamma < |v|$. However, according to Sublemma 8.4, if $\gamma$ was a right-maximal position, we would have $|\widehat{y}| > |w| - |v| \geq 2|v|$. Hence, by Lemma 8.6, there exists a right-maximal position $\delta < \gamma$ with witness $(\overline{x}, \overline{y})$ such that $\widehat{y}$ is a suffix of $\overline{y}$. This configuration is illustrated in Figure 8.11.

  According to Sublemma 8.4, $|\overline{y}| > |w|-|v|$. Let us analyze the length of the common factor $f$ of $x$ and $\overline{y}$. We cannot have $|f| < |v| - 1$, because in that case $\overline{y}$ would start after position $\pi - |v| + 1$. As $\overline{y}$ ends before position $\beta - 1$, this would imply that $|\overline{y}| < 2|v|$, a contradiction with $|\overline{y}| > |w| - |v|$.

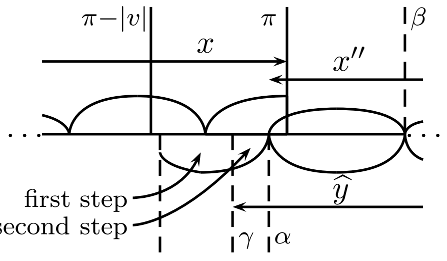

Figure 8.12: Case $|\widehat{y}| \geq \beta - \gamma$.

So Lemma 8.7 can be applied, showing that $x$ and $\overline{y}$ are synchronized. As $\overline{y}$ is a suffix of $w$, and considering the definition of $\gamma$, $\overline{y}$ ends after $\pi$ and allows to extend $x$, in contradiction with the definition of $\pi$.

- Case $|\widehat{y}| \geq \beta - \gamma$ (i.e. $\widehat{y}$ ends at or after position $\beta - 1$).

  We will show that the word $m_{\pi-|v|+2} \cdots m_\pi$ has two distinct $v$-factorizations, which constitutes a contradiction with Lemma 8.7 (as its length is $|v| - 1$). The first $v$-factorization comes from $x$. The second one will be built by extending $x'$ to the left. These two $v$-factorizations are distinct because $\pi$ is a left-maximal position with witness $(x, y)$, so $x$ cannot be extended. In the remainder of the proof, we show how to build the second $v$-factorization from $x'$.

  We proceed in two steps, as described in Figure 8.12:

  - First step. Let $z$ be the common factor between $y$ and $y'$ (we have $|z| = |y|-1$). From the definition of $\gamma$, $z$ appears between positions $\pi-|v|+2$ and $\gamma-1$ (with its two $v$-factorizations as suffix of $y$ and prefix of $y'$): $m_{\pi-|v|+2} \cdots m_{\gamma-1} = z$.
  - Second step. In order to complete the second $v$-factorization (from position $\gamma$ to position $\pi$), we have to get the suffix of $v$ between positions $\gamma$ and $\alpha - 1$. If $\alpha \leq \gamma$, the second $v$-factorization has been completed during the first step. So let us consider that $\gamma < \alpha$. As $|\widehat{y}| \geq \beta - \gamma$ (this corresponds to the second case), $x''$ and $\widehat{y}$ have a common factor of length greater than $|v| - 1$, so by Lemma 8.7 they are synchronized. Hence $\widehat{y}$ enables to extend $x''$ to the left until $\gamma$, and we obtain the second $v$-factorization.

This concludes the proof of Theorem 8.2. □

## 8.6 Family $\mathcal{F}_{\text{p.unb.}}$ of Powers of Unbordered Words

We show that CERTAIN($w$) is aperiodic if $w$ belongs to $\mathcal{F}_{\text{p.unb.}}$, the family of powers of unbordered words (see Definition 8.3).

**Theorem 8.3** *For every word $w \in \mathcal{F}_{p.unb.}$,* CERTAIN($w$) *is aperiodic.*

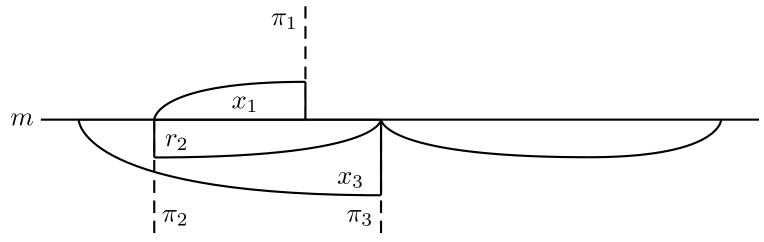

Figure 8.13: Situation in the proof of Sublemma 8.5.

Proof. Let $w = v^h$ with $v$ an unbordered word and $h \geq 1$. The proof is by contradiction, using Lemma 7.1. Let $k$ be sufficiently large (see Remark 8.1). Let $P$, $U$, $Q$ be multiwords such that $PU^kQ \models_{\mathsf{certain}} w$. Assume towards a contradiction that $m = puq$ with $p \in \mathsf{words}(P)$, $u \in \mathsf{words}(U^{k+1})$, and $q \in \mathsf{words}(Q)$ such that $m \not\Vdash w$. Hence the Decomposition Lemma can be applied.

**Sublemma 8.5** *There exist a position $\pi$ in $m$ and an $\ell$-decomposition $(x, y)$ at position $\pi$ such that $|x| \geq |v|$.*

Proof. Assume for contradiction that for every position $\pi$ in $m$ and for every $\ell$-decomposition $(x, y)$ at $\pi$, $|x| < |v|$. For each position $\pi_1$, far enough from the borders of $m$ (see Remark 8.1), there exists an $\ell$-decomposition $(x_1, y_1)$ at position $\pi_1$ by Lemma 8.5. We choose such position $\pi_1$ and $\ell$-decomposition $(x_1, y_1)$ with $x_1$ of maximal length. By our contradiction hypothesis, $|x_1| < |v|$. Let $(x_2, y_2)$ be an $r$-decomposition at position $\pi_2 = \pi_1 - |x_1| + 1$, and consider $r_2$ such that $y_2 \in r_2v^*$ with $0 < |r_2| \leq |v|$. Thus, $r_2$ is a (not necessarily proper) nonempty suffix of $v$. The situation is depicted in Figure 8.13. As $v$ is unbordered, it must be the case that $|r_2| > |x_1|$. Let $(x_3, y_3)$ be an $\ell$-decomposition at position $\pi_3 = \pi_2 + |r_2| - 1$. By our (contradiction) hypothesis, $|x_3| < |v|$. As $v$ is unbordered, it must be the case that $|x_3| > |r_2|$. As $|r_2| > |x_1|$, we have a contradiction with the choice of $\ell$-decomposition $(x_1, y_1)$ at position $\pi_1$ with $x_1$ of maximal length. ⊣

By Sublemma 8.5, we can assume a position $\pi_1$ in $m$ and a prefix $x_1$ of $w$ that ends at position $\pi_1$, and such that $|x_1| \geq |v|$. If $w = v^h$ with $h = 1$, it follows that $x_1 = w$ and thus $m \Vdash w$ which is impossible. When $h \geq 2$, we show in the next paragraph that $x_1m_{\pi_1+1}$ is a prefix of $w$. Then, by repeated application of the same reasoning, we obtain $m \Vdash w$, again a contradiction.

We can assume $j \geq 1$ such that $x_1 = v^jr_1$ with $0 \leq |r_1| < |v|$. Let $(x_2, y_2)$ be an $r$-decomposition at position $\pi_2 = \pi_1 - |r_1| + 1$ where $y_2 \in r_2v^*$ for some $r_2$ satisfying $0 < |r_2| \leq |v|$. Since $v$ is unbordered it must be the case that $|r_2| > |r_1|$. We distinguish two cases:

- Case $|r_2| = |v|$. Obviously, $x_1m_{\pi_1+1}$ is a prefix of $w$.
- Case $|r_2| < |v|$. Let $(x_3, y_3)$ be an $\ell$-decomposition at position $\pi_3 = \pi_2 + |r_2| - 1$. Let $x_3 \in v^*r_3$ for some $r_3$ satisfying $0 < |r_3| \leq |v|$. As $v$ is unbordered, $|r_3| > |r_2|$. It follows that the word $p = m_{\pi_3-|r_3|+1} \cdots m_{\pi_2-1}$ must be a nonempty proper prefix of $v$ (see Figure 8.14). Since $v$ is unbordered, $p$ cannot be a suffix of $v$. Since $x_1 = v^jr_1$

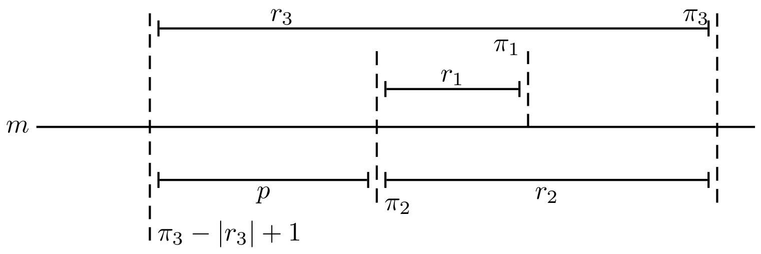

Figure 8.14: Situation in the proof of Theorem 8.3.

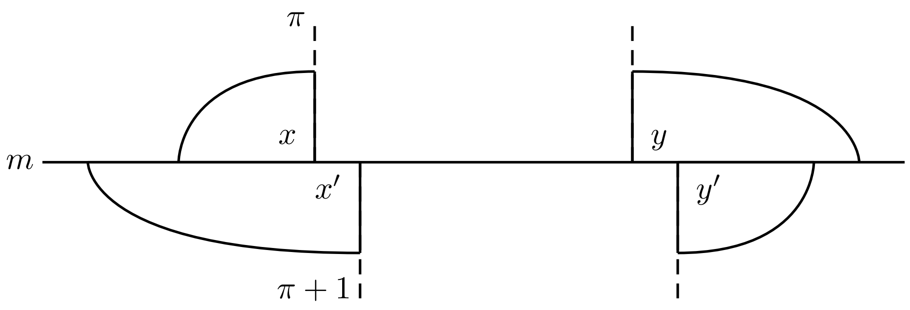

Figure 8.15: Case $j \geq 1$ in the proof of Sublemma 8.6.

is a suffix of $m_1 \cdots m_{\pi_1}$ with $j \geq 1$, we have that $v^j$ is a suffix of $m_1 \cdots m_{\pi_2-1}$. Then, $p$ is a suffix of $v$, a contradiction. We conclude that this case cannot occur.

This concludes the proof.

□

## 8.7 Family $\mathcal{F}_{\text{anch.}}$ of Anchored Words

The third family of words is the family $\mathcal{F}_{\text{anch.}}$ of anchored words as defined in Definition 8.4.

**Theorem 8.4** *For every word $w \in \mathcal{F}_{\text{anch.}}$, CERTAIN$(w)$ is aperiodic.*

Proof. Let $w = savat$ be an anchored word, as in Definition 8.4. The proof is again by contradiction, using Lemma 7.1. Let $k$ be large enough (see Remark 8.1) and let $P, U, Q$ be multiwords such that $PU^kQ \models_{\mathsf{certain}} w$. Assume that there exist $p \in \mathsf{words}(P), u \in \mathsf{words}(U^{k+1}), q \in \mathsf{words}(Q)$ such that $m = puq$ does not contain $w$ as a factor. Therefore Lemma 8.5 can be applied.

**Sublemma 8.6** *There exist a position $\pi$ in $m$ and an $\ell$-decomposition $(x, y)$ at position $\pi$ such that either $x \Vdash ava$ or $y \Vdash ava$.*

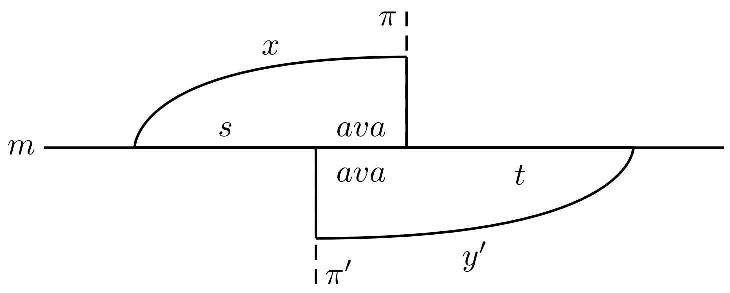

Figure 8.16: Case $x' \Vdash ava$ in the proof of Theorem 8.4.

Proof. Assume the contrary, i.e. for all $\pi$ (far enough from the borders of $w$, see Remark 8.1) and all $\ell$-decompositions $(x, y)$ at position $\pi$, we have $x \nVdash ava$ and $y \nVdash ava$. By Lemma 8.6, we can assume position $\pi$ is left-maximal and its witness $(x, y)$ is such that $x = sav_1 \cdots v_i$ and $y = v_{i+1} \cdots v_n at$ where $n = |v|$. Again, by our assumption, there is an $\ell$-decomposition $(x', y')$ at position $\pi + 1$ such that $x' = sav_1 \cdots v_j$ and $y' = v_{j+1} \cdots v_n at$ for some $j$. If $j \geq 1$, one of $sav_1 \cdots v_i$ or $sav_1 \cdots v_{j-1}$ must be a suffix of the other (see $x, x'$ on Figure 8.15).

We recall that $v$ does not contain the symbol $a$ (by condition (1) of Definition 8.4). It follows that these two words are equal, and therefore $(x, y)$ is not a maximal $\ell$-decomposition, a contradiction. So $j = 0$. Assume $i < |v|$. Considering $y$, the $(|v| - i)$-th symbol of $y'$ must be the symbol $a$, a contradiction as $y' = vat$.

Thus, $j = 0$, $i = |v|$ and $x = sav$, $y = at$, $x' = sa$ and $y' = vat$. Since $m_{\pi+1} = a = m_{\pi+|U|+1}$, it follows that the $\ell$-decomposition $(x, y)$ is not maximal, a contradiction. ⊣

By Sublemma 8.6, there exist a position $\pi$ in $m$ and an $\ell$-decomposition $(x, y)$ at position $\pi$ satisfying $x \Vdash ava$ or $y \Vdash ava$. Suppose $x \Vdash ava$ (the other case is symmetrical). By condition (2) of Definition 8.4, $sava$ is prefix of $x$. It follows that $t \neq \epsilon$, otherwise $m \Vdash w$ which is impossible. We define $\pi' = \pi - |x| + |s| + 1$ (see Figure 8.16). By Lemma 8.5, there is an $r$-decomposition $(x', y')$ at position $\pi'$. By construction, either $ava$ is a proper prefix of $y'$ or $y'$ is prefix of $ava$. Condition (4) implies that only the first case can happen. By condition (2), we must have $y' = avat$. Recall that $sava$ is prefix of $x$. It follows that $w$ is factor of $m$ (see Figure 8.16), a contradiction. □

## 8.8 Family $\mathcal{F}_{\text{unr.}}$ of Unrimmed Words

The last family of words $w$ for which we show the aperiodicity of $\mathsf{CERTAIN}(w)$ is the family $\mathcal{F}_{\text{unr.}}$ of unrimmed words (see Definition 8.5).

We first show that if $w$ is unrimmed, then for every multiword $M$, if $M$ contains a position that is not a singleton, then this position is not relevant to state that $M \in \mathsf{CERTAIN}(w)$. The statement of Lemma 8.8 resembles the one of Lemma 8.1.

**Lemma 8.8** *Let $w$ be an unrimmed word. For every multiword $M$, if $M \in \mathsf{CERTAIN}(w)$ then, for every position $i$ in $M$ such that $|M_i| \geq 2$, we have $M_1 \cdots M_{i-1} \in \mathsf{CERTAIN}(w)$*

*or* $M_{i+1} \cdots M_{|M|} \in \mathsf{CERTAIN}(w)$.

Proof. Let $w$ be an unrimmed word. Assume by contradiction there exist a multiword $M \in \mathsf{CERTAIN}(w)$ and a position $i \in \{1, \ldots, |M|\}$ such that $|M_i| \geq 2$, $M \in \mathsf{CERTAIN}(w)$ but $M_1 \cdots M_{i-1} \notin \mathsf{CERTAIN}(w)$ and $M_{i+1} \cdots M_n \notin \mathsf{CERTAIN}(w)$. We can assume w.l.o.g. words $m_{\text{left}}$ and $m_{\text{right}}$ such that $m_{\text{left}} \in \mathsf{words}(M_1 \cdots M_{i-1})$ and $m_{\text{right}} \in \mathsf{words}(M_{i+1} \cdots M_{|M|})$ such that neither $m_{\text{left}}$ nor $m_{\text{right}}$ contains $w$ as a factor. Since $w$ is a factor of $m_{\text{left}} \cdot a_1 \cdot m_{\text{right}}$ and a factor of $m_{\text{left}} \cdot a_2 \cdot m_{\text{right}}$ with $a_1 \neq a_2$, it must be the case that $w$ has a rim, a contradiction. $\square$

The main result follows.

**Theorem 8.5** *For every word* $w \in \mathcal{F}_{unr.}$, $\mathsf{CERTAIN}(w)$ *is aperiodic.*

Proof. Let $w$ be an unrimmed word. The proof is similar to the one of Theorem 8.1 and relies on Lemma 8.8. Using the same arguments, we have that $\mathsf{CERTAIN}(w)$ can be expressed by the following first-order sentence:

$$\exists i_1 \exists i_2 \ldots \exists i_{|w|} \Big( S(i_1, i_2) \wedge \ldots S(i_{|w|-1}, i_{|w|}) \wedge P_{\{w_1\}}(i_1) \wedge P_{\{w_2\}}(i_2) \wedge \cdots \wedge P_{\{w_{|w|}\}}(i_{|w|}) \Big).$$

It follows by Theorem 7.1 that $\mathsf{CERTAIN}(w)$ is aperiodic. $\square$

The first-order sentence in the proof of Theorem 8.5 explains the first-order sentence of Example 6.5:

**Example 8.1** Let $w = aba$ be the word of Example 6.5, an unrimmed word. A multiword $M$ belongs to $\mathsf{CERTAIN}(w)$ if $\mathsf{struct}(M)$ satisfies the following formula.

$$\exists i \exists j \exists k \big( S(i,j) \wedge S(j,k) \wedge P_{\{a\}}(i) \wedge P_{\{b\}}(j) \wedge P_{\{a\}}(k) \big).$$

$\triangleleft$

## 8.9 Coverage of the Families of Words

Aperiodicity was easy to show for partial words defined on an alphabet with at least three symbols. Somewhat surprisingly, aperiodicity proofs turn out to be much harder for multiwords where uncertain positions can contain exactly two symbols.

It is still an open conjecture that $\mathsf{CERTAIN}(w)$ is aperiodic (and hence first-order definable) for any word $w$. This conjecture has been experimentally verified on a very large set of (about $80,000,000$) words for different sizes of alphabet, including all words of length less than or equal to 12 over $\Sigma = \{a, b, c\}$.

Those experiments were conducted by means of a set of tools we developed. Those tools are publicly available on the Launchpad platform[2]. The tools use AMoRE[3] to check aperiodicity.

[2]See Multiwords Project at `https://launchpad.net/multiwords/`.

[3]AMoRE (for computing Automata, MOnoids, and Regular Expressions) is a tool developed by Christian Albrechts at the University of Kiel. The tool can be found at `http://amore.sourceforge.net/`.

Using different techniques, we showed in this chapter the aperiodicity of $\mathsf{CERTAIN}(w)$ for a large number of words $w$. Our proofs strongly rely on some synchronization properties of such words, and these techniques do not extend to arbitrary words.

We conducted experiments to study the size of each of those families for several fixed lengths of words and several fixed sizes of the alphabet. The situation strongly differs when we consider an alphabet of size two and alphabets of larger sizes.

Figure 8.17 shows the coverage of each of the four families of words for words of length 1 to 16 over an alphabet of two symbols while Figure 8.18 shows the results for an alphabet of three symbols. On those two figures, the curve labelled "*others*" indicates the words that do not belong to any of the four families. Note that there exist words that belong to more than one family and thus they can be counted more than once.

The following observations can be made.

1. The family $\mathcal{F}_{\text{p.unb.}}$ of powers of unbordered words is a large family. Over an alphabet of two symbols, it is the largest considered family of words. Over an alphabet of three symbols, for $5 \leq |w| \leq 14$, it is the second one and still contains more than one word out of two.

2. There is a large proportion of words over an alphabet of three symbols that are anchored. For instance, more than 90% of the words of length 14 over $\{a, b, c\}$ are anchored.

3. The family $\mathcal{F}_{\text{rep.3}}$ is the smallest family, either in the case of an alphabet of two symbols or of three symbols.

4. Over an alphabet of two symbols, the proportion of words that do not belong to one of the four families increases with the length of the words.

5. The situation completely differs for an alphabet of three symbols: the proportion of words outside our families decreases when the length of the considered words increases. For words of length $\geq 11$, the proportion is less than or equal to 10%.

We give some concrete figures. Let $\Sigma = \{a, b, c\}$ and let $L$ contain all words $w$ over $\Sigma$ such that $1 \leq L \leq 14$. We have the following.

| Language | Language size | Percentage |
|---|---|---|
| $L \cap \mathcal{F}_{\text{rep.3}}$ | 450 | 0.006% |
| $L \cap \mathcal{F}_{\text{p.unb.}}$ | 3,999,906 | 55.8% |
| $L \cap \mathcal{F}_{\text{anch.}}$ | 6,445,509 | 89.8% |
| $L \cap \mathcal{F}_{\text{unr.}}$ | 747,654 | 10.4% |
| $L \cap$ others | 464,754 | 6.5% |
| $L$ | 7,174,452 | 100% |

It turns out that many words are anchored. Figure 8.19 represents the proportion of the different families for alphabets of two to six symbols. Note that the length of the words is fixed to 8. The trends observed for an alphabet of three symbols are confirmed

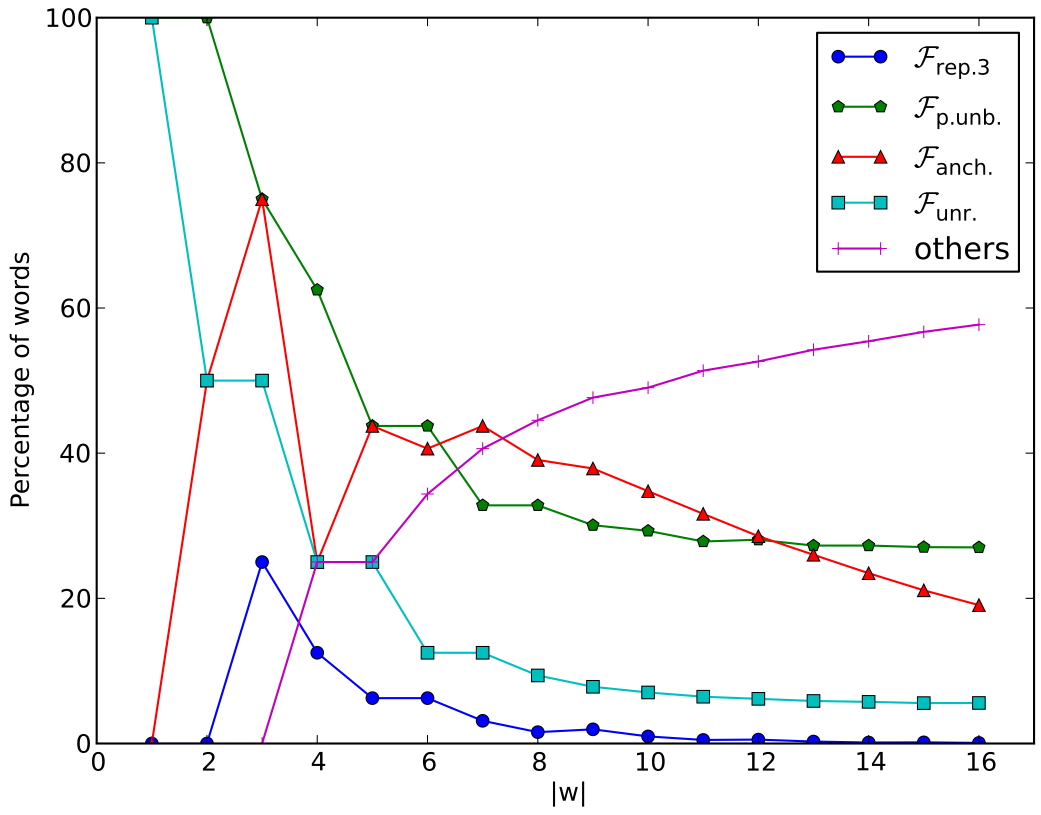

Figure 8.17: Percentage of words $w$ over alphabet $\{a, b\}$ that belong to one of the families for $|w| \leq 16$.

for alphabets of four or more symbols. The families $\mathcal{F}_{\text{anch.}}$ of anchored words and $\mathcal{F}_{\text{p.unb.}}$ of powers of unbordered words are strongly dominant. The proportion of words that do not belong to any of the families decreases both when the length of the words or the number of symbols in the alphabet increases.

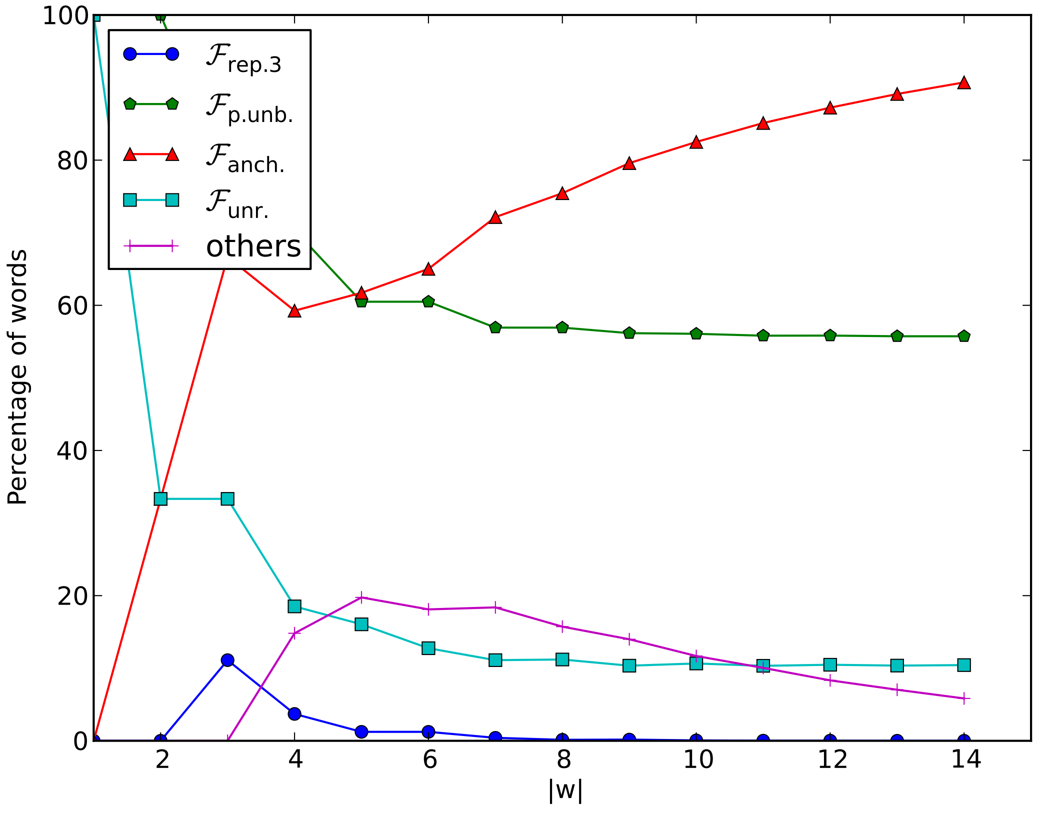

Figure 8.18: Percentage of words $w$ over alphabet $\{a, b, c\}$ that belong to one of the families for $|w| \leq 14$.

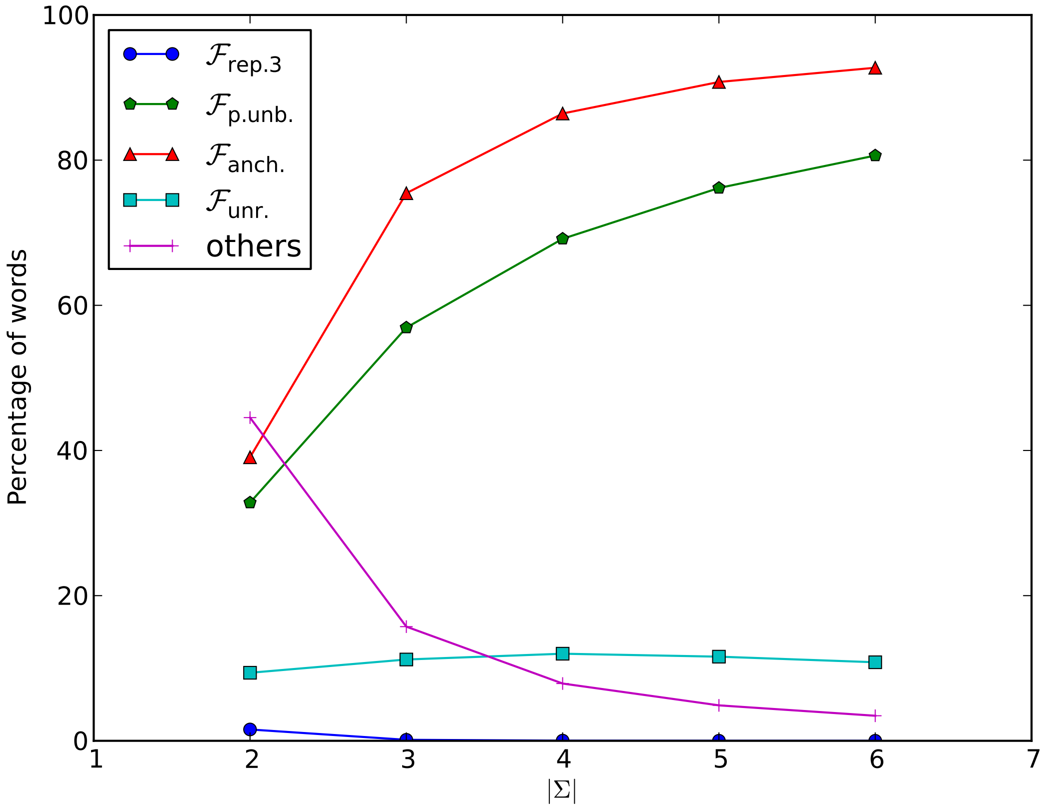

Figure 8.19: Percentage of words $w$ of fixed length 8 that belong to one of the families. The size of the alphabet ranges from 2 to 6.

CHAPTER 9

# Automata Recognizing CERTAIN($w$)

The previous chapter provided sufficient conditions for aperiodicity of CERTAIN($w$). This chapter studies CERTAIN($w$) and CERTAIN$_\diamond$($w$) from an automaton perspective. The motivation is that automata theory could give some insights in the algebraic structure of the languages CERTAIN($w$), for any word $w$.

Section 9.1 recalls the notations and terminologies used in automata theory. This section is mainly based on [Knuth et al. 1977; Hopcroft et al. 2007]. In Section 9.2, we give a procedure for the construction of a minimal deterministic finite state automaton that recognizes CERTAIN$_\diamond$($w$). In Section 9.3, we explain how to construct a deterministic finite automaton for CERTAIN($w$) for every word $w$.

We developed a set of tools that, given a word $w$, automatically computes a minimal (in terms of number of states) automaton that recognizes the language CERTAIN($w$). In Section 9.4, we experimentally study the size of these minimal automata. We identify potential upper and lower bounds on the number of states of those minimal automata. Sections 9.4.1 and 9.4.2 focus on families of words $w$ whose minimal automata reach one of those bounds.

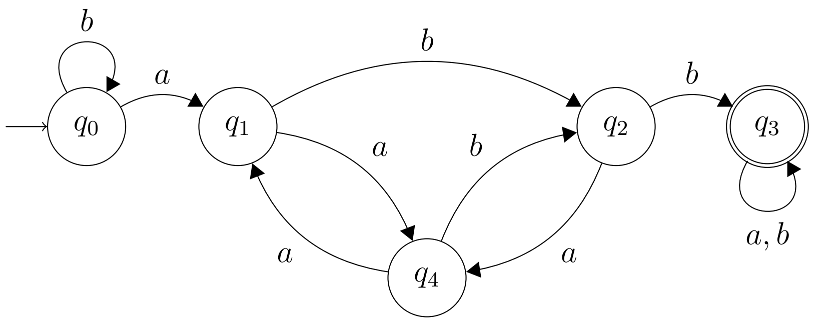

Figure 9.1: Example of a DFA over $\Sigma = \{a, b\}$.

## 9.1 Preliminaries

A *deterministic finite automaton* (DFA) $D$ over an alphabet $\Sigma$ is described by a 5-tuple: $D = (Q, \Sigma, q_0, F, \delta)$ where $Q$ is a set of states, $q_0 \in Q$ the initial state, $F \subseteq Q$ a set of final states and $\delta : Q \times \Sigma \to Q$ is the *transition function.* It is usual to represent an automaton using a graph. From a graph perspective, if $\delta(q, a) = p$ for some $p \in Q, q \in Q, a \in \Sigma$, then there's an arc from state $q$ to state $p$ labelled by $a$.

Let $w = w_1 w_2 \cdots w_n$ be a sequence of letters from $\Sigma$. The DFA starts in its initial state $q_0$ by looking at $\delta(q_0, w_1) = q_1$ to find the state the DFA enters after processing the first letter $w_1$. The next state is obtained by computing $\delta(q_1, w_2) = q_2$, and so on until $w_n$ is read. Assume $\delta(q_{n-1}, w_n) = q_n$. If $q_n \in F$, then $q_n$ is called a *final state* and $w$ is *accepted.* If $q_n \notin F$, then $w$ is *rejected.* The language recognized by a DFA is the set of all words that this DFA accepts.

If $\delta$ is a transition function, then we define $\delta^* : Q \times \Sigma^* \to Q$ as its extension to words. The definition is inductive, as follows:

- $\delta^*(q, \epsilon) = q$ (basis);
- $\delta^*(q, w) = \delta(\delta^*(q, x), a)$ for $w = x \cdot a$ (induction).

A state $q$ is *reachable* if there exists some input word $w$ such that $\delta^*(q_0, w) = q$ for $q_0$ the initial state. Obviously, a word $w$ is accepted by the DFA if $\delta^*(q_0, w) \in F$.

**Example 9.1** Let $D = (Q, \Sigma, q_0, F, \delta)$ be the DFA shown in Figure 9.1. We have $\Sigma = \{a, b\}$ and $Q = \{q_0, q_1, q_2, q_3, q_4\}$. We have $F = \{q_3\}$, the set of final states, and the initial state is $q_0$. This deterministic finite automaton recognizes the language of words that have $abb$ as a factor. $\triangleleft$

A DFA $D$ recognizing a language $L$ is called *minimal* if, for any DFA $D'$ recognizing the same language $L$, the number of states of $D$ is smaller or equal to the number of states of $D'$. Given $L$, a minimal automaton recognizing $L$ is unique up to a renaming of the states. The minimization of an automaton is effective (see Chapter 4 in [Hopcroft et al. 2007]) and will be summarized in Section 9.3.

For example, the automaton shown in Figure 9.1 has 5 states. This automaton is not minimal. States $q_1$ and $q_4$ are equivalent because their *residual languages* are identical: on

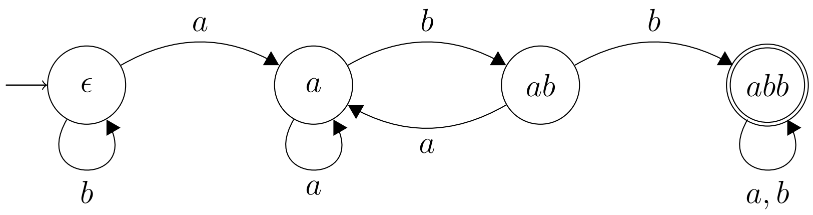

Figure 9.2: Construction of the minimal automaton $\mathsf{KMP}(abb)$.

every input word $w$, $\delta^*(q_1, w)$ is an accepting state if and only if $\delta^*(q_4, w)$ is an accepting state. If we merge $q_1$ and $q_4$, we obtain a smaller automaton that accepts the same language.

Given a word $w$, there exists a procedure to directly construct a minimal DFA that recognizes the language of words that have $w$ as a factor. This procedure is based on the Knuth-Morris-Pratt algorithm and is described in Definition 9.1. The procedure always results in a DFA which has $|w| + 1$ states and which is minimal [Crochemore et al. 2007].

**Definition 9.1** [Aho et al. 1974] Let $w$ be a word. We define the following deterministic finite automaton $\mathsf{KMP}(w)$. Let $\mathsf{KMP}(w) = (Q, \Sigma, q_0, F, \delta)$ where

- the set of states $Q$ is $\{\epsilon, w_1, w_1w_2, w_1w_2w_3, \dots, w\}$;
- $\Sigma$ is the alphabet;
- the initial state $q_0$ is $\epsilon$;
- the set of final states $F = \{w\}$;
- if $p$ is a prefix of $w$ and $a$ a letter from $\Sigma$, then
  - $\delta(p, a) = w$ if $p = w$, or
  - $\delta(p, a) = q$ where $q$ is the longest suffix of $p \cdot a$ which is also a prefix of $w$.

$\triangleleft$

The construction of the transition function in Definition 9.1 is quite similar to the construction of the shift function in the Knuth-Morris-Pratt algorithm of Section 7.2. Figure 9.2 shows the resulting automaton $\mathsf{KMP}(abb)$ for word $abb$. Thus, the automata of Figure 9.1 and 9.2 accept the same language.

## 9.2 Automata Recognizing $\mathsf{CERTAIN}_\diamond(w)$

In this section, for an alphabet $\Sigma$ with $|\Sigma| \geq 3$, we consider the construction of a deterministic finite automaton that recognizes the language $\mathsf{CERTAIN}_\diamond(w)$. Recall from Section 6.2 (1) that a multiword $M = M_1M_2 \dots M_n$ is a partial word if each $M_i$ is either a singleton or the entire alphabet, and (2) that $\mathsf{CERTAIN}_\diamond(w)$ is the set of all partial words in which $w$ is certain.

We show in the following definition a deterministic finite automaton $\mathcal{A}_\diamond(w)$ whose construction is quite similar to the construction of $\mathsf{KMP}(w)$ of Definition 9.1. The transitions of $\mathcal{A}_\diamond(w)$ are roughly the same as in $\mathsf{KMP}(w)$ except for the transitions that are labelled by a symbol that is not a singleton. These transitions lead to the initial state.

**Definition 9.2** Let $w$ be a word. Let $\mathsf{KMP}(w) = (Q, \Sigma, q_0, F, \delta')$. We define the deterministic finite automaton $\mathcal{A}_\diamond(w) = (Q, \widehat{\Sigma}, q_0, F, \delta)$ where $\widehat{\Sigma} = 2^\Sigma \setminus \{\emptyset\}$. The transition function $\delta$ is defined next. For every $a \in \Sigma$,

- $\delta(f, a) = f$, if $f \in F$;
- $\delta(q, a) = \delta'(q, a)$ if $a$ is a singleton and $q \notin F$;
- $\delta(q, a) = q_0$ if $a$ is not a singleton and $q \notin F$.

$\lhd$

We show in Theorem 9.1 that $\mathcal{A}_\diamond(w)$ recognizes $\mathsf{CERTAIN}_\diamond(w)$ and is minimal.

**Theorem 9.1** *Let $\Sigma$ be an alphabet of at least three symbols. Let $w$ be a word. Then $\mathcal{A}_\diamond(w)$ recognizes $\mathsf{CERTAIN}_\diamond(w)$ and is minimal.*

Proof. Let $\Sigma$ be an alphabet of at least three symbols. Let $w$ be a word and $M$ a partial word. Let $\mathsf{KMP}(w)$ and $\mathcal{A}_\diamond(w)$ be as in Definition 9.2. Let $\delta$ be the transition function of $\mathcal{A}_\diamond(w)$ and $\delta^*$ be its extension. We show $M \in \mathsf{CERTAIN}_\diamond(w) \iff \delta^*(q_0, M) \in F$. The proof is by induction on the number $n$ of positions $i$ of $M$ such that $M_i$ is not a singleton.

- Case $n = 0$. If $M$ contains only singletons, then a run of $\mathcal{A}_\diamond(w)$ on $M$ behaves exactly as a run of $\mathsf{KMP}(w)$ on $m$ where $m$ is the only word of $\mathsf{words}(M)$.
- Case $n \geq 1$. Let $M = PAQ$ for suitable multiwords $P$ and $Q$, where $A \in \widehat{\Sigma}$ such that $|A| = |\Sigma| \geq 3$ and $Q$ is only composed of symbols that are singletons.

  $\boxed{\Rightarrow}$ Assume $M \in \mathsf{CERTAIN}_\diamond(w)$. As $|A| \geq 3$, we can apply Lemma 8.1 and either $P \in \mathsf{CERTAIN}_\diamond(w)$ or $Q \in \mathsf{CERTAIN}_\diamond(w)$. We distinguish two cases:
  1. $P \in \mathsf{CERTAIN}_\diamond(w)$. Then, by induction, $P$ is accepted by $\mathcal{A}_\diamond(w)$ and $\delta^*(q_0, P) \in F$.
  2. $P \notin \mathsf{CERTAIN}_\diamond(w)$. Then $Q \in \mathsf{CERTAIN}_\diamond(w)$. We have $\delta^*(q_0, P) = q \notin F$, $\delta(q, A) = q_0$. As $Q$ only contains singletons, it follows $\delta^*(q_0, Q) \in F$.

  We have that $\mathcal{A}_\diamond(w)$ accepts $M$.

  $\boxed{\Leftarrow}$ Assume $\mathcal{A}_\diamond(w)$ accepts $M$. We distinguish two cases:
  1. $\delta^*(q_0, P) \in F$. This implies $P \in \mathsf{CERTAIN}_\diamond(w)$ and so $M \in \mathsf{CERTAIN}_\diamond(w)$.
  2. $\delta^*(q_0, P) = q \notin F$. By construction of $\mathcal{A}_\diamond(w)$, we have $\delta(q, A) = q_0$. As $\mathcal{A}_\diamond(w)$ accepts $M$, it must be the case that $\delta^*(q_0, Q) \in F$. But $Q$ contains only singletons, so we have $Q \in \mathsf{CERTAIN}_\diamond(w)$ and thus $M \in \mathsf{CERTAIN}_\diamond(w)$.

It follows that $\mathcal{A}_\diamond(w)$ recognizes $\mathsf{CERTAIN}_\diamond(w)$. It is easy to see that $\mathcal{A}_\diamond(w)$ has $|w| + 1$ states and is minimal. $\square$

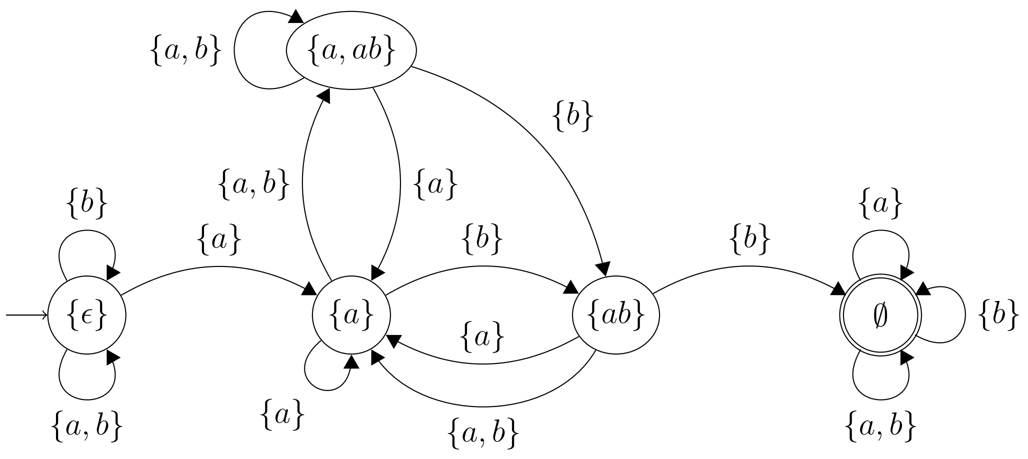

Figure 9.3: The automaton $\mathcal{A}(abb)$ over $\Sigma = \{a, b\}$.

## 9.3 Automata Recognizing $\mathsf{CERTAIN}(w)$

Theorem 7.2 on page 88 suggests the following construction of a deterministic finite automaton for $\mathsf{CERTAIN}(w)$.

**Definition 9.3** Let $P$ be the set of proper prefixes of $w$. We define the deterministic finite automaton $\mathcal{A}(w) = (Q, \widehat{\Sigma}, S_0, F, \delta)$ on the powerset alphabet $\widehat{\Sigma}$.

- The finite set of states is $Q = \{\lfloor S \rfloor \mid S \subseteq P\}$.
- The initial state is $S_0 = \{\epsilon\}$, and the final states are $F = \{\emptyset\}$.
- The transition function $\delta : Q \times \widehat{\Sigma} \to Q$ is defined[1] by:

$$\delta(S, A) = \lfloor \mathsf{sufpre}(S \cdot A, w) \setminus \{w\} \rfloor$$

◁

For instance, Figure 9.3 shows $\mathcal{A}(abb)$ for the alphabet $\Sigma = \{a, b\}$. Theorem 7.2 and the construction of $\mathcal{A}(\cdot)$ immediately lead to the following result.

**Theorem 9.2** *For every word $w$, the automaton $\mathcal{A}(w)$ recognizes* $\mathsf{CERTAIN}(w)$.

It is important to notice that, for every word $w$, the automaton $\mathcal{A}(w)$ present a lot of similarities with the automaton $\mathsf{KMP}(w)$ introduced in Definition 9.1. In particular, a run of $\mathsf{KMP}(w)$ on an input $a_1 a_2 \cdots a_n$ behaves exactly as a run of $\mathcal{A}(w)$ on the multiword $\{a_1\}\{a_2\} \cdots \{a_n\}$. This is not surprising as the transition functions are defined similarly: if $p$ is a state and $a$ the letter to be read, these functions both compute the longest suffix

[1]Recall from Section 7.2 that, if $S$ is a set of words, $\lfloor S \rfloor$ is the smallest set of words satisfying $\lfloor S \rfloor \subseteq S$ and $\lfloor S \rfloor$ contains a suffix of every word in $S$. If $p, q$ are words, $\mathsf{sufpre}(p, q)$ is the longest suffix of $p$ that is a prefix of $q$.

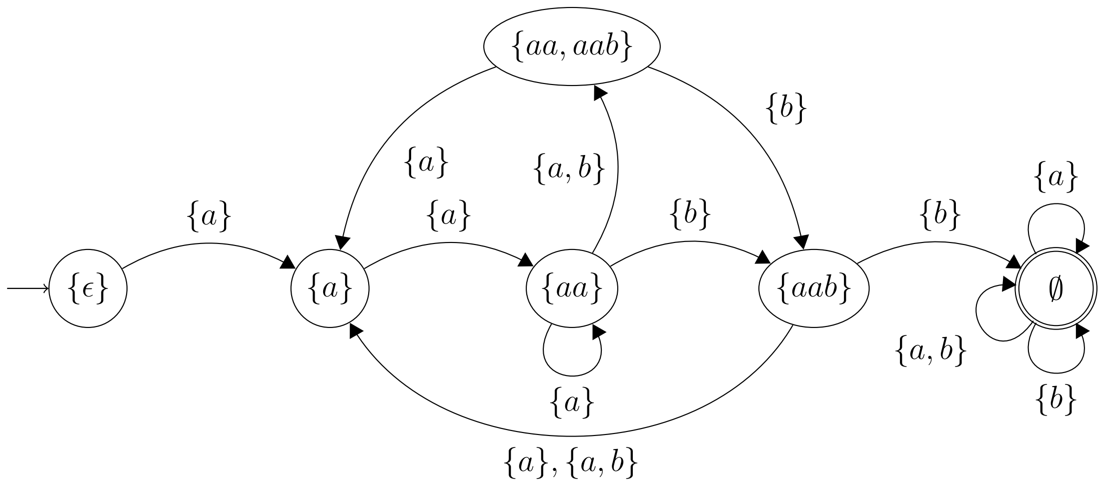

Figure 9.4: A minimal automaton recognizing CERTAIN($aabb$). The transitions leading to the initial state $\{\epsilon\}$ are not shown for clarity.

of $p \cdot a$ that is also a prefix of $w$. The difference in the computation comes from the fact that the construction of $\mathcal{A}(w)$ works on subsets of prefixes of $w$ and not on prefixes of $w$ as in KMP($w$).

Unfortunately, while the construction of KMP($w$) always leads to a minimal automaton whose size is exactly $|w| + 1$ states, the construction of $\mathcal{A}(w)$ does not necessarily result in a minimal automaton. For example, the automaton of Figure 9.3 is not minimal: states $\{a, b\}$ and $\{a\}$ can be merged, resulting in a minimal automaton with $|w| + 1$ states. Importantly, there exist words $w$ such that a minimal automaton for CERTAIN($w$) has more than $|w| + 1$ states. For example, the automaton of Figure 9.4 recognizes CERTAIN($w$) where $w = aabb$. This DFA is minimal but has 6 ($= |w| + 2$) states.

If $w$ is a word, $\mathcal{A}_{\min}(w)$ denotes the minimal automaton recognizing CERTAIN($w$). The minimization of an automaton is an effective procedure, i.e. one can compute $\mathcal{A}_{\min}(w)$ from $\mathcal{A}(w)$. The minimization procedure is described in [Hopcroft et al. 2007] and can be summarized as follows:

1. Remove any state that cannot be reached from the initial state.

2. Partition the remaining states into blocks such that two states in the same blocks are equivalent. Two states $p$ and $q$ are equivalent if, for every word $w$, $\delta^*(p, w)$ is a final state if and only if $\delta^*(q, w)$ is a final state.

The resulting automaton is a minimal automaton recognizing the same language. The study of minimal automata for CERTAIN($w$) may provide useful insights, in particular in the aperiodicity of CERTAIN($w$). For this purpose, we developed a set of tools that automatically construct $\mathcal{A}_{\min}(w)$. In the next section, we explain the experiments we conducted to study the size of the minimal automata for CERTAIN($w$).

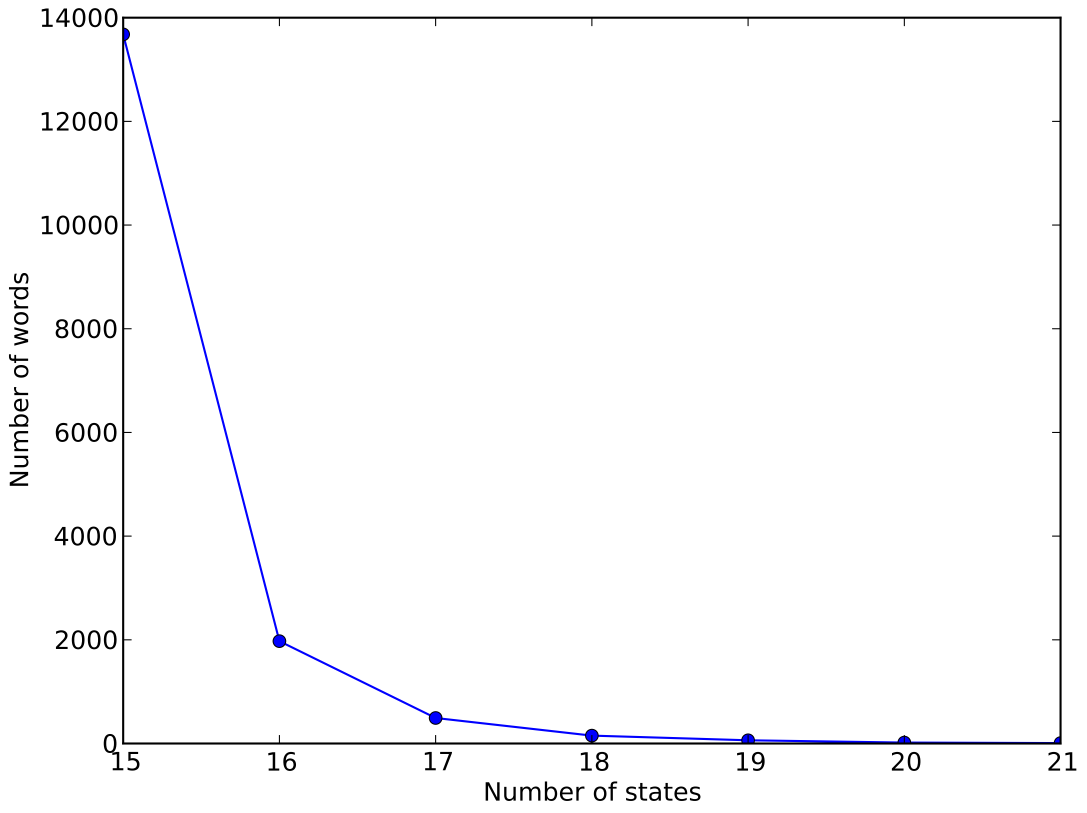

Figure 9.5: Let $\Sigma = \{a, b\}$. For every word $w$ such that $|w| = 14$, the number of states of $\mathcal{A}_{\min}(w)$ is in the range $[15, 21]$. For every $n$ in this range, the graph shows the number of words $w$ such that $|w| = 14$ and $\mathcal{A}_{\min}(w)$ has $n$ states.

## 9.4 Number of States of $\mathcal{A}_{\min}(w)$

We computed $\mathcal{A}_{\min}(w)$ for a large number of words $w$ in order to study its number of states. For alphabet $\Sigma = \{a, b\}$, we computed the size of $\mathcal{A}_{\min}(w)$ for all words $w$ such that $|w| \leq 16$. For larger words, we computed the size of $\mathcal{A}_{\min}(w)$ for randomly generated $w$. The experiments were conducted using our set of tools which use python-automata library[2] and GAP[3]. The tools are publicly available on the Launchpad platform[4].

The first experiments concern an exhaustive study of the set of words $w$ over $\Sigma = \{a, b\}$ such that $2 \leq |w| \leq 16$. The following observations are striking.

1. No minimal automaton $\mathcal{A}_{\min}(w)$ has more than $|w| + \lfloor \frac{|w|}{2} \rfloor$ states for $|w| \geq 2$.

[2]python-automata is a Python library to handle basic automata operations, developed by Andrew Badr, and is available at `https://code.google.com/p/python-automata/`.

[3]GAP (Groups, Algorithms, Programming) is a system for computational discrete algebra which is available at `http://www.gap-system.org/`.

[4]See Multiwords Project at `https://launchpad.net/multiwords/`.

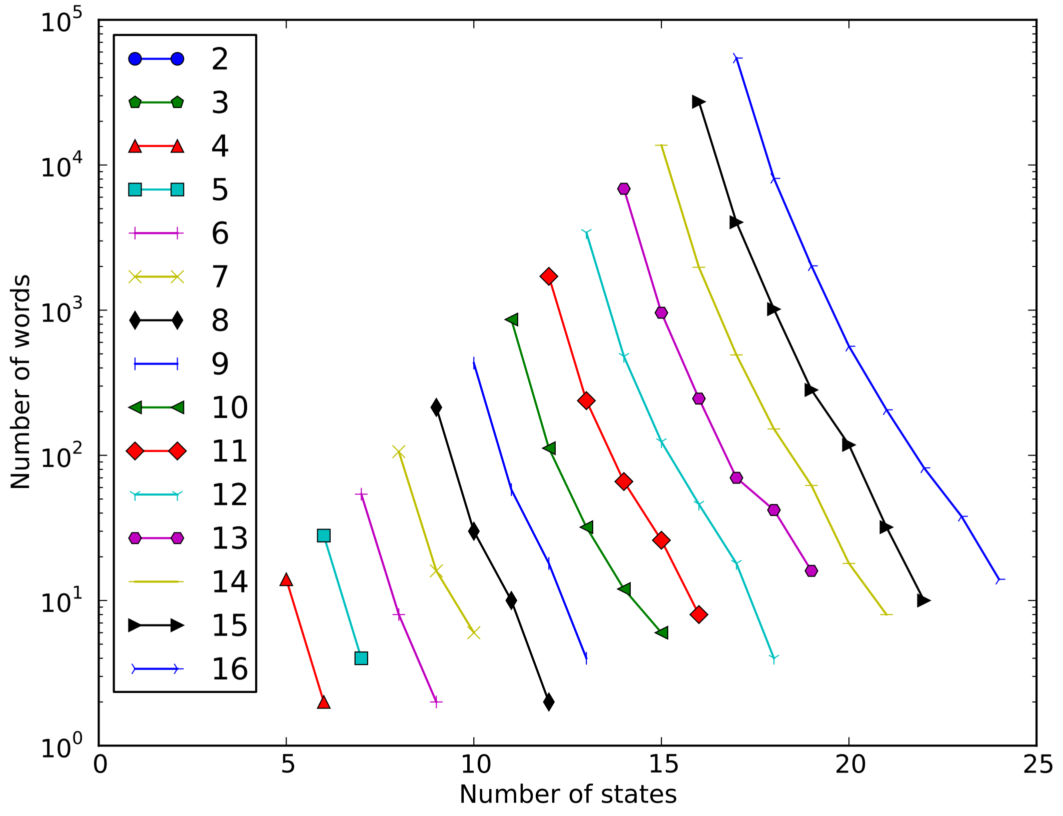

Figure 9.6: Let $\Sigma = \{a, b\}$. For every word $w$ such that $2 \leq |w| \leq 16$, the number of states of $\mathcal{A}_{\min}(w)$ is in the range $[|w| + 1, |w| + \lfloor \frac{|w|}{2} \rfloor]$. The graph with number $n$ shows, for every $k$ in this range, the number of words $w$ such that $|w| = n$ and $\mathcal{A}_{\min}(w)$ has $k$ states. The ordinate scale is logarithmic.

2. For a fixed length $n$, there is a majority of words of length $n$ such that $\mathcal{A}_{\min}(w)$ has $n+1$ states, as in $\mathsf{KMP}(w)$. If $\mathcal{A}_{\min}(w)$ has $n+1$ states, then this means that each state of $\mathcal{A}(w)$ having more than one prefix can be merged with a state having only one prefix, i.e. a state (equivalent to a state) of the Knuth-Morris-Pratt automaton.

   The situation is depicted in Figure 9.5, for all words of length 14. For these words, the size of $\mathcal{A}_{\min}(w)$ ranges from 15 to 21. For $n \in \{15, \dots, 21\}$, the graph shows the number of words $w$ such that $\mathcal{A}_{\min}(w)$ has $n$ states.

3. The proportion of words $w$ such that $\mathcal{A}_{\min}(w)$ has $|w|+1$ states seems to be similar for all word lengths. For $\Sigma = \{a, b\}$ and words $w$ such that $6 \leq |w| \leq 16$, this proportion is approximately 83%. This is Figure 9.6. The number of states of $\mathcal{A}_{\min}(w)$ is in the range $[|w|+1, |w| + \lfloor\frac{|w|}{2}\rfloor]$. The ordinate scale is logarithmic.

We also studied the impact of the alphabet size on the size of $\mathcal{A}_{\min}(w)$. We computed the number of states of $\mathcal{A}_{\min}(w)$ for words $w$ of fixed length 8 for alphabet sizes ranging from 2 to 5. The results are shown in Figure 9.7 and do not contradict the three observations above.

A non-exhaustive search for randomly generated words over $\{a, b\}$ of length up to 28 confirmed those observations for words of greater length. We computed the size of $\mathcal{A}_{\min}(w)$ for about $430,000$ words and never got a counter-example of the three observations made. Figure 9.8 shows the number of words $w$ in function of the number of states of $\mathcal{A}_{\min}(w)$. The length of $w$ ranges from 17 to 28. There are about $110,000$ words $w$ such that $17 \leq |w| \leq 24$ and about $320,000$ words $w$ such that $25 \leq |w| \leq 26$. Data are represented in the same way as in Figure 9.6 and confirm the observations we made.

Based on these observations, we conjecture an upper bound of $|w| + \lfloor\frac{|w|}{2}\rfloor$ on the number of states of $\mathcal{A}_{\min}(w)$.

Importantly, we do not see any obvious link between the fact that a word $w$ belongs to one of the families described in the previous chapter and the number of states of $\mathcal{A}_{\min}(w)$, with the notable exception of unrimmed words $w$ which have a minimal DFA with $|w|+1$ states, as stated in Proposition 9.1 of the following subsection. Table 9.1 shows that, within the same family of words, the number of states of $\mathcal{A}_{\min}(w)$ can vary. In the two following subsections, we focus on families of words $w$ for which $\mathcal{A}_{\min}(w)$ has exactly $|w|+1$ states or exactly $|w| + \lfloor\frac{|w|}{2}\rfloor$ states.

### 9.4.1 Families of words $w$ minimizing $|\mathcal{A}_{\min}(w)|$

One of the observations made during the exhaustive search is that, for a large majority of words $w$, a minimal automaton for $\mathsf{CERTAIN}(w)$ has exactly $|w|+1$ states, the same number of states as the minimal automaton $\mathsf{KMP}(w)$ of Definition 9.1.

The following proposition states that for every word $w \in \mathcal{F}_{\text{unr.}}$, $\mathsf{CERTAIN}(w)$ can be recognized by an automaton with $|w|+1$ states. Remember that this family includes the well known family of palindromes.

**Proposition 9.1** *If $w$ is an unrimmed word, then a minimal deterministic automaton recognizing* $\mathsf{CERTAIN}(w)$ *has $|w|+1$ states.*

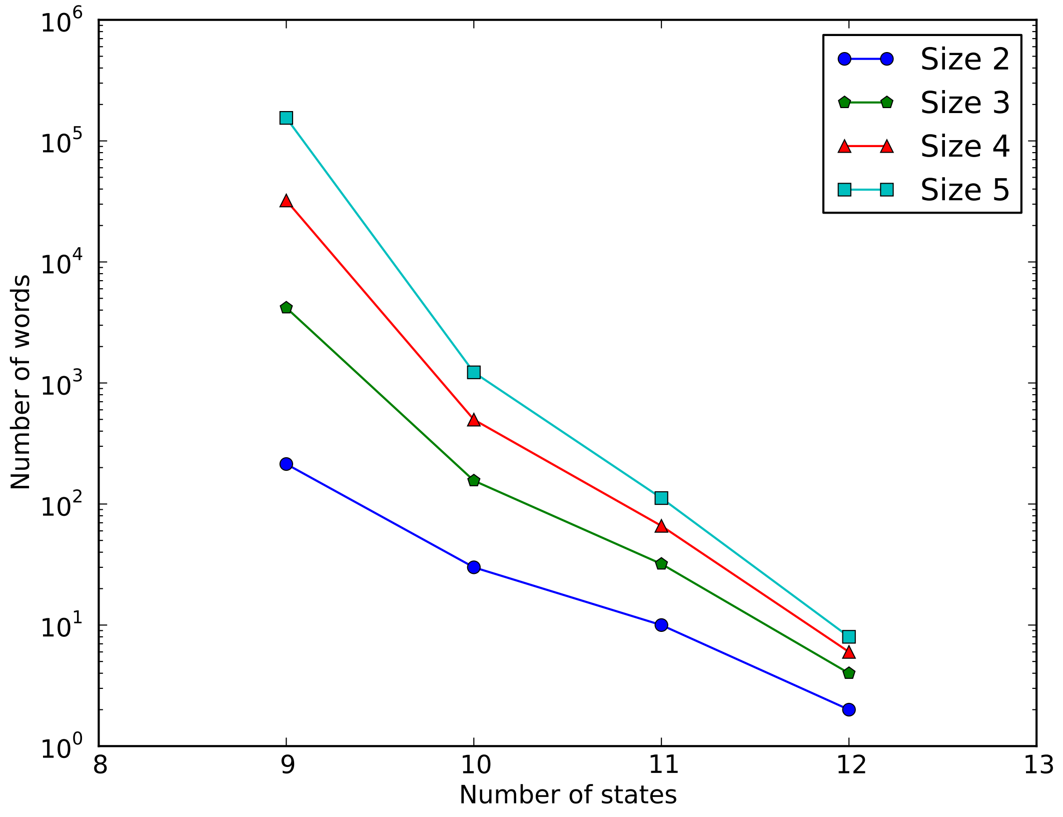

Figure 9.7: For every word $w$ with $|w| = 8$, the number of states of $\mathcal{A}_{\text{min}}(w)$ is in the range $[9, 12]$. The graph with label "Size $i$" concerns an alphabet $\Sigma$ such that $|\Sigma| = i$. It shows, for $n \in [9, 12]$, the number of words $w$ of length 8 such that $\mathcal{A}_{\text{min}}(w)$ has $n$ states. Notice that the graph labeled "size 2" is the same as the graph labeled "8" in Figure 9.6.

| $w$ | $\lvert w\rvert$ | $\lvert\mathcal{A}_{\text{min}}(w)\rvert$ | $\mathcal{F}_{\text{rep.3}}$ | $\mathcal{F}_{\text{p.unb.}}$ | $\mathcal{F}_{\text{anch.}}$ |
|---|---|---|---|---|---|
| $(abaa)^3$ | 12 | $12+1$ | $\times$ | | |
| $(aabb)^3$ | 12 | $12+2$ | $\times$ | $\times$ | |
| $abbb$ | 4 | $4+1$ | | $\times$ | |
| $aaabb$ | 5 | $5+2$ | | $\times$ | $\times$ |
| $aaabab$ | 6 | $6+3$ | | $\times$ | $\times$ |
| $abbaba$ | 6 | $6+1$ | | | $\times$ |
| $abaaa$ | 5 | $5+1$ | | | |
| $aaababa$ | 7 | $7+3$ | | | |

Table 9.1: The number of states of $\mathcal{A}_{\text{min}}(w)$ can vary for words $w$ in the same family.

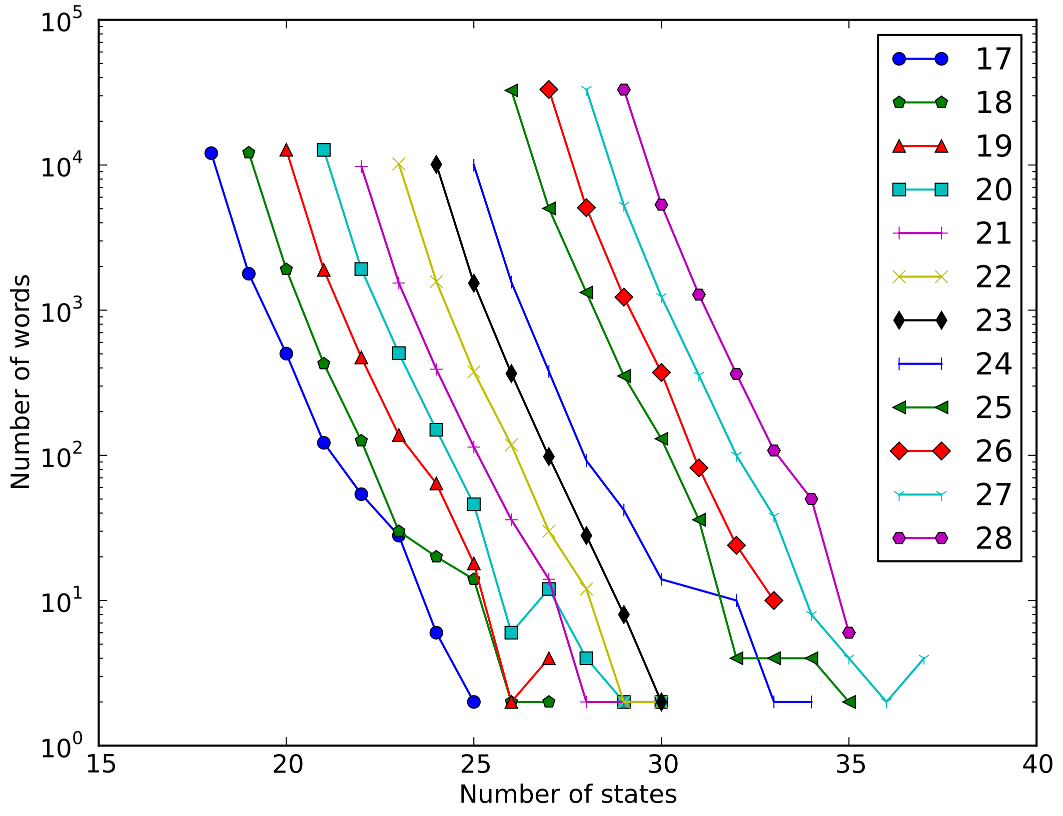

Figure 9.8: For randomly generated words $w$ such that $17 \leq |w| \leq 28$, the number of states of $\mathcal{A}_{\min}(w)$ is in the range $[|w|+1, |w| + \lfloor\frac{|w|}{2}\rfloor]$. The graph with number $n$ shows, for every $k$ in this range, the number of words $w$ such that $|w| = n$ and $\mathcal{A}_{\min}(w)$ has $k$ states. The ordinate scale is logarithmic.

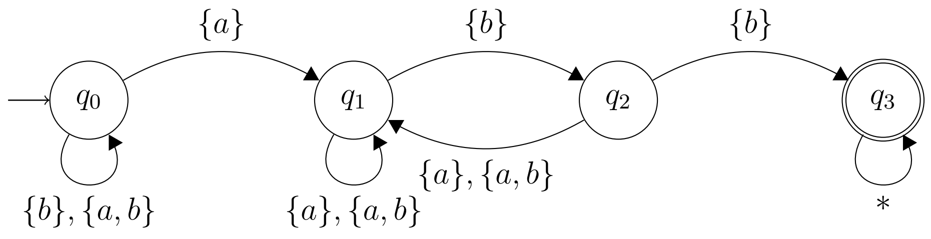

Figure 9.9: A minimal automaton for CERTAIN($abb$).

Proof. Let $w$ be an unrimmed word. One can see that $\mathcal{A}_\diamond(w)$ of Definition 9.2 recognizes CERTAIN($w$). The proof relies on Lemma 8.8 which states that, for every multiword $M$, if $M$ has a position $i$ such that $M_i$ is not a singleton, then this position $i$ is not relevant to state that $w$ is certain in $M$.

The proof is similar to the one of Theorem 9.1 and, using the same arguments, it follows that $\mathcal{A}_\diamond(w)$ recognizes CERTAIN($w$), has $|w| + 1$ states and is minimal. □

The particularity of $\mathcal{A}_\diamond(w)$ is that every transition labelled by a set of at least two letters systematically leads to the initial state, except for the transitions that come from a final state. The converse of Proposition 9.1 is not true. For instance, a minimal DFA for CERTAIN($abb$) has 4 states but $abb$ has two rims. Figure 9.9 shows a minimal automaton recognizing CERTAIN($abb$). Notice that $\mathcal{A}_{\min}(abb)$ contains two transitions labelled by $\{a, b\}$ that do not lead to the initial state.

## 9.4.2 Families of words $w$ maximizing $|\mathcal{A}_{\min}(w)|$

Through the experiments, we identified several families of words $w$ for which a minimal automaton $\mathcal{A}_{\min}(w)$ has $|w| + \lfloor \frac{|w|}{2} \rfloor$. These families are:

- $a^{k+2}ba^kb$ (for $0 \leq k$)
- $a^{k+2}ba^jba^kba^jb$ (for $0 < j < k$)
- $a^{k+3}ba^kb$ (for $0 \leq k$)
- $a^{k+3}ba^jba^kba^jb$ ($0 < j < k$)

For $w = a^{k+2}ba^kb$, we explicitly construct a minimal automaton $\mathcal{A}_{\min}(w)$ and we show that it has a size of $3k + 6 = |w| + \lfloor \frac{|w|}{2} \rfloor$.

For the construction of this automaton, we apply the procedure described in Definition 9.3 and we minimize the resulting automaton. Figure 9.10 shows the automaton $\mathcal{A}(w)$ for $w = a^{k+2}ba^kb$, depending on $k$. A minimal automaton is obtained by removing the state in gray (equivalent to state $\{a^{k+2}, a^{k+2}b\}$). Notice that for simplicity, transitions leading to state $\{\epsilon\}$ have been omitted.

**Proposition 9.2** *Let $k \geq 0$. Let $w = a^{k+2}ba^kb$. A minimal automaton that recognizes* $\mathsf{CERTAIN}(w)$ *has $|w| + \lfloor \frac{|w|}{2} \rfloor$ states.*

Proof. Let $w = a^{k+2}ba^kb$ with $k \geq 0$. We use the procedure described in Definition 9.3 to construct the automaton $\mathcal{A}(w)$. The resulting automaton is illustrated in Figure 9.10. Notice that for simplicity, all the transitions that lead to state $\{\epsilon\}$ and the non-reachable states have been omitted in the figure. The procedure ensures that this automaton recognizes exactly the language $\mathsf{CERTAIN}(w)$. For this automaton, we say that $q$ is a *singleton* state if it contains at most one prefix, otherwise $q$ is said to be a *composed* state.

It is easy to see that the state in gray is equivalent to the state $\{a^{k+2}, a^{k+2}b\}$ so it can be merged with it. It can also be shown that the remaining states cannot be merged. To show that two states cannot be merged, it suffices to exhibit words that belong to the residual language of the first state but not to the residual language of the second state.

We now show that the remaining states are not equivalent and thus, the automaton is minimal and has exactly $|w| + \lfloor \frac{|w|}{2} \rfloor$ states. By contradiction, suppose there exist two different states $q_1$ and $q_2$ such that $q_1$ and $q_2$ are equivalent. We distinguish four cases:

1. $q_1$ and $q_2$ are singleton states;
2. $q_1$ and $q_2$ are composed states (i.e. states that contain two or more prefixes);
3. $q_1$ is a composed state, $q_2$ is a singleton state with a prefix $p$ such that $|p| > k + 2$;
4. $q_1$ is a composed state, $q_2$ is a singleton state with a prefix $p$ such that $|p| \leq k + 2$.

Let $\delta$ be the transition function and $\delta^*$ be its extension for words (see Section 9.1). For each case, it suffices to show that there exists a multiword $M$ such that $\delta^*(q_1, M)$ and $\delta^*(q_2, M)$ are not equivalent. Notice that, for simplicity, if $M$ is a multiword, we write $M = v$ as a shortcut for $M = \{v_1\}\{v_2\} \cdots \{v_{|v|}\}$.

**Case 1** Let $q_1$ and $q_2$ be singleton states. We can assume that $q_1$ (resp. $q_2$) contains some prefix $p_1$ (resp. $p_2$) such that $|p_1| < |p_2|$. Let $M = s_2$ for some suitable $s_2$ such that $w = p_2 \cdot s_2$. Clearly, $\delta^*(q_2, s_2) = \emptyset \neq \delta^*(q_1, s_2)$.

**Case 2** Let $q_1$ and $q_2$ be composed states. We can assume that $q_1$ (resp. $q_2$) contains prefixes $a^{k+2}$ and $a^{k+2}ba^i$ (resp. $a^{k+2}$ and $a^{k+2}ba^j$) such that $0 \leq i < j \leq k$. Let $M = a^{k-j}ba^kb$. It is easy to see that $\delta^*(q_2, M) = \emptyset$ and $\delta^*(q_1, M) = \{\epsilon\}$.

**Case 3** Let $q_1$ be a composed state containing prefixes $a^{k+2}$ and $a^{k+2}ba^i$ $(0 \leq i \leq k)$ and $q_2$ a singleton state containing $a^{k+2}ba^j$ $(0 \leq j \leq k)$. Let $M = a^{k-j}b$. Clearly, $\delta^*(q_2, M) = \emptyset \neq \delta^*(q_1, M)$.

**Case 4** Let $q_1$ be a composed state containing prefixes $a^{k+2}$ and $a^{k+2}ba^i$ $(0 \leq i \leq k)$ and $q_2$ a singleton state containing $a^j$ $(0 \leq j \leq k + 2)$. Three cases occur:

- $i = j + 2$. Let $M = a^{k-i}a$. Then $\delta^*(q_1, M) = \{a^{k+1}\}$ and $\delta^*(q_2, M) = \{a^{k+2}\}$. By case 1, these states are not equivalent;

- $i < j + 2$. Let $M = a^{k-i}ba^kb$. Then $\delta^*(q_1, M) = \emptyset$ and $\delta^*(q_2, M) = \{\epsilon\}$;
- $i > j + 2$. Let $M = a^{k-j+2}ba^kb$. Then $\delta^*(q_1, M) = \{\epsilon\}$ and $\delta^*(q_2, M) = \emptyset$.

This concludes the proof. □

In this chapter, we studied deterministic finite automata for $\mathsf{CERTAIN}(w)$ and for $\mathsf{CERTAIN}_\diamond(w)$. We provided a procedure to construct a minimal deterministic finite automaton for $\mathsf{CERTAIN}_\diamond(w)$ if the alphabet has at least three symbols. We also provided a procedure to construct a DFA for $\mathsf{CERTAIN}(w)$. We discussed the relationship between this automaton and $\mathsf{KMP}(w)$. We conducted several experiments to compute the (size of) minimal automata for $\mathsf{CERTAIN}(w)$. These experiments resulted in the following intriguing conjecture.

**Conjecture 9.1** *Let $\Sigma$ be an alphabet such that $|\Sigma| \geq 2$. For every nonempty word $w \in \Sigma^*$, the number of states in $\mathcal{A}_{min}(w)$ is smaller than or equal to $\frac{3|w|}{2}$.*

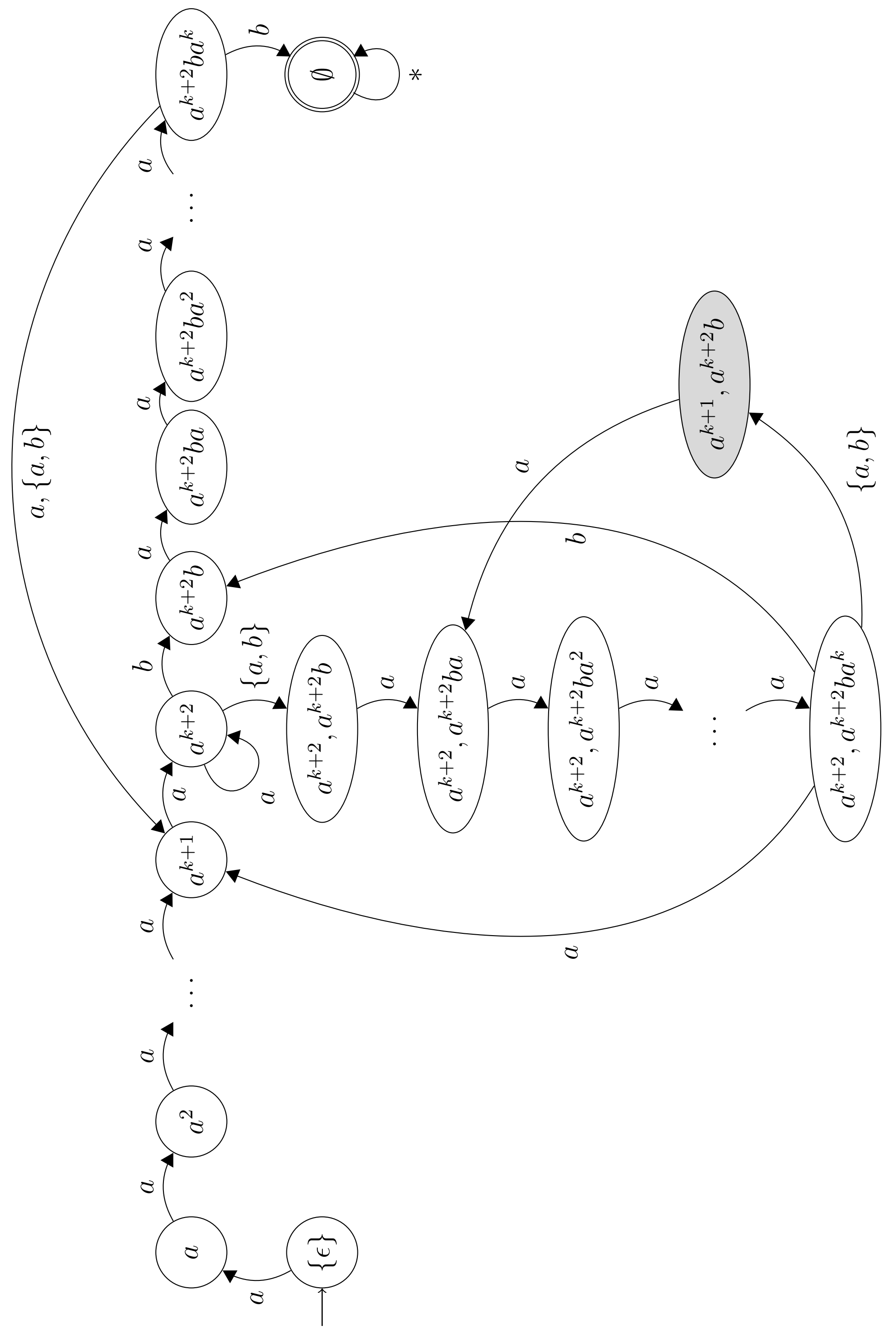

Figure 9.10: Automaton $\mathcal{A}(w)$ for $w = a^{k+2}ba^kb$. Transitions leading to $\{\epsilon\}$ and non-reachable states are not shown for clarity.

CHAPTER 10

# Conclusions

In the framework of first-order logic on words, uncertainty is captured by the concept of multiword, which is a finite sequence of nonempty sets of possible symbols. Motivated by a problem in uncertain databases, we studied the first-order definability of $\mathsf{CERTAIN}(w)$. A multiword $M$ is in $\mathsf{CERTAIN}(w)$ if the word $w$ is certain in $M$, this is, if $w$ is a factor of every word represented by $M$. We studied the conjecture that $\mathsf{CERTAIN}(w)$ is first-order definable for every word $w$. As $\mathsf{CERTAIN}(w)$ is regular, the study of its first-order definability is equivalent to the study of its aperiodicity.

The aperiodicity of $\mathsf{CERTAIN}(w)$ is easy to show if $\mathsf{CERTAIN}(w)$ is restricted to partial words over an alphabet of at least three symbols, but turns out to be difficult in the general case. We provided strong evidence to support the conjecture that $\mathsf{CERTAIN}(w)$ is first-order definable for every word $w$.

We showed aperiodicity of $\mathsf{CERTAIN}(w)$ for words in the following classes: repeated ($\geq$ 3) words, powers of unbordered words, anchored words and unrimmed words. We showed that those families are incomparable under set inclusion and we studied the coverage of each of them. We observed that the proportion of words outside these families decreases when the length of the words or the size of the alphabet increases.

We experimentally verified aperiodicity of $\mathsf{CERTAIN}(w)$ on large sets of words, including all words of length less than or equal to 12 over $\Sigma = \{a, b, c\}$.

We studied the set $\mathsf{CERTAIN}(w)$ from an automaton perspective, expecting strong algebraic properties that could be used to show the aperiodicity of $\mathsf{CERTAIN}(w)$ in the general case. We gave a procedure for the construction of a deterministic finite state automaton recognizing $\mathsf{CERTAIN}(w)$. In particular, we were interested in the number of states of the minimal automaton recognizing $\mathsf{CERTAIN}(w)$ and $\mathsf{CERTAIN}_{\diamond}(w)$, the restriction of $\mathsf{CERTAIN}(w)$ to partial words.

We showed that, for every word $w$ over an alphabet with three or more symbols, there exists an automaton recognizing $\mathsf{CERTAIN}_{\diamond}(w)$ with $|w| + 1$ states. The automaton is minimal and can be effectively constructed. If the alphabet contains less than three symbols, $\mathsf{CERTAIN}_{\diamond}(w)$ is equal to $\mathsf{CERTAIN}(w)$. The study of a minimal automaton for $\mathsf{CERTAIN}(w)$ turns out to be more difficult. We experimentally studied the size of those minimal automata. We computed the size of $\mathcal{A}_{\min}(w)$ for large sets of words, including every word $w$ such that $|w| \leq 16$ over $\Sigma = \{a, b\}$. Our experiments indicate that for every word $w$, a minimal automaton for $\mathsf{CERTAIN}(w)$ has a number of states between $|w| + 1$ and $\frac{3|w|}{2}$. We formally showed that there exist words $w$ such that $\mathsf{CERTAIN}(w)$ cannot be recognized by an automaton with less than $\lfloor \frac{3|w|}{2} \rfloor$ states.

Our study reveals several problems for future research. It is still an open conjecture that $\mathsf{CERTAIN}(w)$ is aperiodic and thus first-order definable for every word $w$. The minimal deterministic finite automata recognizing $\mathsf{CERTAIN}(w)$ exhibit several similarities with the Knuth-Morris-Pratt automata. A better comprehension of a minimal automaton for $\mathsf{CERTAIN}(w)$ could give some useful insight to study the first-order definability of $\mathsf{CERTAIN}(w)$. In particular, it could be useful to identify which properties on $w$ imply that $\mathcal{A}_{\text{min}}(w)$ has more than $|w| + 1$ states. It could be interesting to study other aspects than the number of states of those automata as, for example, the number of *forward* and *backward* transitions (see [Crochemore et al. 2007]).

The study of $\mathsf{CERTAIN}(w)$ can also be extended to words with variables. For example, a pattern $xx$ where $x$ is a variable is certain in the multiword $M = a\{a, b\}\{a, b\}\{a, b\}b$ because every word in $\mathsf{words}(M)$ contains $aa$ or $bb$ as a factor. Note that $\mathsf{CERTAIN}(xx) \neq \big(\mathsf{CERTAIN}(aa) \cup \mathsf{CERTAIN}(bb)\big)$. The conjecture that $\mathsf{CERTAIN}(w)$ is first-order definable does not extend to words with variables. It can been seen that multiword $a\{a, b\}^i b$ is in $\mathsf{CERTAIN}(xx)$ if and only if $i$ is odd. The latter condition is not first-order definable.

# Bibliography

Abiteboul S., Hull R., and Vianu V. 1995. *Foundations of Databases.* Addison-Wesley

Afrati F.N. and Kolaitis P.G. 2009. Repair checking in inconsistent databases: algorithms and complexity. In R. Fagin, ed., *ICDT*, volume 361 of *ACM International Conference Proceeding Series*, pp. 31–41. ACM

Aho A.V. and Corasick M.J. 1975. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340

Aho A.V., Hopcroft J.E., and Ullman J.D. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley

Arenas M., Bertossi L., and Chomicki J. 2003. Answer sets for consistent query answering in inconsistent databases. *Theory Pract. Log. Program.*, 3(4):393–424

Arenas M., Bertossi L.E., and Chomicki J. 1999. Consistent query answers in inconsistent databases. In *PODS*, pp. 68–79. ACM Press

Beeri C., Fagin R., Maier D., and Yannakakis M. 1983. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513

Berstel J. and Boasson L. 1999. Partial words and a theorem of Fine and Wilf. *Theoretical Computer Science*, 218(1):135–141

Bertossi L. 2006. Consistent query answering in databases. *SIGMOD Rec.*, 35(2):68–76

Bertossi L.E. 2011. *Database Repairing and Consistent Query Answering.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers

Bertossi L.E., Bravo L., Franconi E., and Lopatenko A. 2008. The complexity and approximation of fixing numerical attributes in databases under integrity constraints. *Inf. Syst.*, 33(4-5):407–434

Bienvenu M. 2012. Inconsistency-tolerant conjunctive query answering for simple ontologies. In Y. Kazakov, D. Lembo, and F. Wolter, eds., *Description Logics*, volume 846 of *CEUR Workshop Proceedings*. CEUR-WS.org

Blanchet-Sadri F. 2007. *Algorithmic Combinatorics on Partial Words (Discrete Mathematics and Its Applications).* Chapman & Hall/CRC

Boyer R.S. and Moore J.S. 1977. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772

Bruyère V., Carton O., Decan A., Gauwin O., and Wijsen J. 2012. An aperiodicity problem for multiwords. *RAIRO – Theoretical Informatics and Applications*, 46:33–50

Bruyère V., Decan A., and Wijsen J. 2009. On first-order query rewriting for incomplete database histories. In *Proceedings of the 2009 16th International Symposium on Temporal Representation and Reasoning*, TIME '09, pp. 54–61. IEEE Computer Society, Washington, DC, USA

Büchi J.R. 1960. Weak second-order arithmetic and finite automata. *Z. Math. Logik und Grundl. Math.*, 6:66–92

Calì A. 2005. Query answering by rewriting in glav data integration systems under constraints. In *Proceedings of the Second international conference on Semantic Web and Databases*, SWDB'04, pp. 167–184. Springer-Verlag, Berlin, Heidelberg

Celle A. and Bertossi L.E. 2000. Querying inconsistent databases: Algorithms and implementation. In *Proceedings of the First International Conference on Computational Logic*, CL '00, pp. 942–956. Springer-Verlag, London, UK

Chomicki J. and Marcinkowski J. 2004. Inconsistency tolerance. Chapter On the computational complexity of minimal-change integrity maintenance in relational databases, pp. 119–150. Springer-Verlag, Berlin, Heidelberg

Chomicki J. and Marcinkowski J. 2005. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.*, 197(1-2):90–121

Chomicki J., Marcinkowski J., and Staworko S. 2004. Computing consistent query answers using conflict hypergraphs. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, CIKM '04, pp. 417–426. ACM, New York, NY, USA

Codd E.F. 1969. Derivability, redundancy and consistency of relations stored in large data banks. *IBM Research Report, San Jose, California*, RJ599

Crochemore M., Hancart C., and Lecroq T. 2007. *Algorithms on Strings.* Cambridge University Press. 392 pages

Crochemore M. and Rytter W. 1994. *Text Algorithms.* Oxford University Press

Dalvi N.N., Ré C., and Suciu D. 2009. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94

Dalvi N.N., Re C., and Suciu D. 2011. Queries and materialized views on probabilistic databases. *J. Comput. Syst. Sci.*, 77(3):473–490

DALVI N.N. and SUCIU D. 2007. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4):523–544

DECAN A., PIJCKE F., and WIJSEN J. 2012. Certain conjunctive query answering in SQL. In E. Hüllermeier, S. Link, T. Fober, and B. Seeger, eds., *Scalable Uncertainty Management*, volume 7520 of *Lecture Notes in Computer Science*, pp. 154–167. Springer Berlin Heidelberg

FINE N.J. and WILF H.S. 1965. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16:109–114

FISCHER M. and PATERSON M. 1974. String matching and other products. *SIAM-AMS Proceedings, Complexity of Computation*, 7:113–125

FONTAINE G. 2013. Why it is hard to obtain a dichotomy conjecture for consistent query answering. In *Proceedings of Logic In Computer Science (LICS 2013)*

FUXMAN A., FAZLI E., and MILLER R.J. 2005. Conquer: Efficient management of inconsistent databases. In F. Özcan, ed., *SIGMOD Conference*, pp. 155–166. ACM

FUXMAN A. and MILLER R.J. 2005. First-order query rewriting for inconsistent databases. In T. Eiter and L. Libkin, eds., *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pp. 337–351. Springer

FUXMAN A. and MILLER R.J. 2007. First-order query rewriting for inconsistent databases. *Journal of Computer and System Sciences*, 73(4):610–635

GOTTLOB G., LEONE N., and SCARCELLO F. 2002. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627

HALAVA V., HARJU T., and KÄRKI T. 2007. Relational codes of words. *Theoretical Computer Science*, 389(1-2):237–249

HOLUB J., SMYTH W.F., and WANG S. 2008. Fast pattern-matching on indeterminate strings. *Journal of Discrete Algorithms*, 6(1):37–50

HOPCROFT J.E., MOTWANI R., and ULLMAN J.D. 2007. *Introduction to Automata Theory, Languages and Computation.* Pearson Addison-Wesley, Upper Saddle River, NJ, 3. edition

KNUTH D.E., MORRIS J.H., and PRATT V.R. 1977. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350

KOLAITIS P.G. and PEMA E. 2012. A dichotomy in the complexity of consistent query answering for queries with two atoms. *Inf. Process. Lett.*, 112(3):77–85

KUCHEROV G., NOÉ L., and ROYTBERG M.A. 2007. Subset seed automaton. In *Proceedings of the 12th International Conference on Implementation and Application of Automata (CIAA)*, pp. 180–191. Springer

Libkin L. 2004. *Elements of Finite Model Theory.* Springer

Lopatenko A. and Bertossi L. 2006. Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. In *Proceedings of the 11th international conference on Database Theory*, ICDT'07, pp. 179–193. Springer-Verlag, Berlin, Heidelberg

Lothaire M. 1997. *Combinatorics on words.* Cambridge University Press

Maier D. 1983. *The Theory of Relational Databases.* Computer Science Press

Maslowski D. and Wijsen J. 2011. On counting database repairs. In G.H.L. Fletcher and S. Staworko, eds., *LID*, pp. 15–22. ACM

Maslowski D. and Wijsen J. 2013. A dichotomy in the complexity of counting database repairs. *J. Comput. Syst. Sci.*

McNaughton R. and Papert S. 1971. *Counter-free Automata.* MIT Press, Cambridge, MA

Papadimitriou C.H. 1994. *Computational complexity.* Addison-Wesley

Pin J.É. 1986. *Varieties of Formal Languages.* North Oxford, London and Plenum, New-York

Rahman M.S., Iliopoulos C.S., and Mouchard L. 2007. Pattern matching in degenerate DNA/RNA sequences. In M. Kaykobad and M.S. Rahman, eds., *Workshop on Algorithms and Computation (WALCOM)*, pp. 109–120. Bangladesh Academy of Sciences (BAS)

Reiter R. 1982. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling (Intervale)*, pp. 191–233

Schützenberger M.P. 1965. On finite monoids having only trivial subgroups. *Information and Control*, 8(2):190–194

Staworko S. and Chomicki J. 2010. Consistent query answers in the presence of universal constraints. *Inf. Syst.*, 35(1):1–22

Tuzhilin A. and Clifford J. 1990. A temporal relational algebra as a basis for temporal relational completeness. In *Proceedings of the sixteenth international conference on Very large databases*, pp. 13–23. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

Ullman J.D. 1988. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press

Wijsen J. 2005. Database repairing using updates. *ACM Trans. Database Syst.*, 30(3):722–768

Wijsen J. 2009a. Consistent query answering under primary keys: a characterization of tractable queries. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pp. 42–52. ACM, New York, NY, USA

Wijsen J. 2009b. On the consistent rewriting of conjunctive queries under primary key constraints. *Inf. Syst.*, 34(7):578–601

Wijsen J. 2010. On the first-order expressibility of computing certain answers to conjunctive queries over uncertain databases. In J. Paredaens and D.V. Gucht, eds., *PODS*, pp. 179–190. ACM

Wijsen J. 2012. Certain conjunctive query answering in first-order logic. *ACM Trans. Database Syst.*, 37(2):9:1–9:35

# List of Figures

# List of Tables