

# MATILDA: Mining Approximate Tuple-Generating Dependencies in Large Databases

FRANÇOIS AMAT, Télécom Paris & Institut Polytechnique de Paris, France

PIERRE-HENRI PARIS, LISN, Paris-Saclay University, CNRS, France

PAOLO PAPOTTI, EURECOM, France

FABIAN M. SUCHANEK, Télécom Paris & Institut Polytechnique de Paris, France

First-Order Tuple-generating dependencies (TGDs) are general constraints on databases that subsume Horn rules, foreign key constraints, and inclusion dependencies. We introduce an approach to automatically mine these dependencies from a given database, subject to a number of constraints. Our method employs tailored pruning techniques and a graph representation to deal with the exponential size of the search space. It can mine both exact and approximate rules by employing support and confidence metrics. Experiments on various benchmark datasets demonstrate that (1) our approach can find all inclusion dependencies and Horn rules that competing approaches find, (2) it scales to databases that are out of reach for competing approaches, and (3) it finds a small portion of more complex patterns (TGDs with multiple atoms in the rule head) that other methods cannot find by design. Overall, MATILDA mines about 50% more joinable patterns than competing systems across 35 benchmark databases while keeping runtimes competitive (sub-second to a few minutes depending on the database size).

## ACM Reference Format:

François Amat, Pierre-Henri Paris, Paolo Papotti, and Fabian M. Suchanek. 2026. MATILDA: Mining Approximate Tuple-Generating Dependencies in Large Databases. 1, 1 (April 2026), 26 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 Introduction

A database consists of one or more tables, each with labeled columns and rows; a toy example about parents and children is shown in Figure 1. In this paper, we are interested in *patterns in the data*. In our toy example, several such patterns emerge:

**Inclusion dependencies** such as:

“Every married person has a child”  
 $\forall p_1, p_2 : \text{Marriage}(p_1, p_2) \Rightarrow \exists c : \text{Child}(p_1, c)$

**Horn rules** such as:

“If a person is married to someone who has a child, then this person is also a parent of that child”  
 $\forall p_1, p_2, c : \text{Marriage}(p_1, p_2) \wedge \text{Lineage}(p_2, c) \Rightarrow \text{Lineage}(p_1, c)$

---

Authors' Contact Information: François Amat, Télécom Paris & Institut Polytechnique de Paris, France, [francois.amat@telecom-paris.fr](mailto:francois.amat@telecom-paris.fr); Pierre-Henri Paris, LISN, Paris-Saclay University, CNRS, France, [pierre-henri.paris@universite-paris-saclay.fr](mailto:pierre-henri.paris@universite-paris-saclay.fr); Paolo Papotti, EURECOM, France, [paolo.papotti@eurecom.fr](mailto:paolo.papotti@eurecom.fr); Fabian M. Suchanek, Télécom Paris & Institut Polytechnique de Paris, France, [fabian.suchanek@telecom-paris.fr](mailto:fabian.suchanek@telecom-paris.fr).

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM XXXX-XXXX/2026/4-ART

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Table MARRIAGE		Table LINEAGE		Table RESIDENCE		
Partner1	Partner2	Parent	Child	Person	Location	State
Elvis	Priscilla	Priscilla	Lisa	Elvis	Memphis	TN
Priscilla	Elvis	Elvis	Lisa	Priscilla	Memphis	TN
Lisa	Danny	Lisa	Riley	Lisa	Beverly Hills	CA
Danny	Lisa	Danny	Riley			

Fig. 1. A Simple Relational Database.

**Tuple-Generating Dependencies (TGDs)** such as:

“Spouses live in the same place”

$$\forall p_1, p_2 : \text{Marriage}(p_1, p_2) \Rightarrow \exists l, s : \text{Residence}(p_1, l, s) \wedge \text{Residence}(p_2, l, s)$$

TGDs generalize Horn rules and inclusion dependencies. These patterns can also be approximate. In our example, Lisa and Danny are married, but one of them does not have a residence. Thus, the pattern holds only in a certain percentage of cases. Such approximate patterns carry insights about real-world correlations. For instance, an HR department may observe that people in senior positions are mostly married, an insurance company is interested in finding which car brands are most involved in accidents, and a scientist may want to know whether all measurements in a particular scenario appear in another scenario as well [63]. Such patterns are also useful in schema evolution [109, 114], data integration [13], AI explainability [7, 71], design rule validation [64], and the augmentation of Large Language Models with structured knowledge [67]. TGDs, in particular, are needed, e.g., for data exchange [83] and ontology-mediated query answering [37].

Several methods aim to mine such patterns automatically. However, most existing methods focus either on specific rule types, such as inclusion dependencies [4, 68, 72] or are poorly applicable to databases, either because they are conceived for knowledge bases [45, 59, 93] or because they are conceived for smaller datasets, as it is typically the case for Inductive Logic Programming (ILP) [11]. As we show in our experiments, there is currently no approach that can mine Horn rules (let alone TGDs) from databases with more than a few thousand rows.

That is not surprising, as mining such patterns faces two main challenges:

- (1) The **search space** is enormous: each new variable, table, or existential quantifier potentially increases the number of candidate rules exponentially – and even more so if we include approximate TGDs.
- (2) Two syntactically different TGDs can be **semantically equivalent**, which does not just inflate the search space but also requires duplicate elimination.

In this paper, we introduce **MATILDA**<sup>1</sup>, the first method that automatically discovers *approximate joinable First-Order TGDs* in large relational databases. We overcome the challenges of pattern mining by targeting *joinable TGDs*, i.e., TGDs whose variables can be linked through equality joins on attributes that appear in at least two relations (as, e.g., in foreign-key–like relationships). We also exclude arithmetic relations.

These design decisions, paired with specialized metrics and smart pruning, drastically reduce the search space and enable MATILDA to scale to millions of tuples. We further introduce a novel constraint graph representation that not only reduces duplicates but also significantly prunes the space of possible patterns.

<sup>1</sup>Mining Approximate TGDs In Large Databases

Our experimental evaluation against traditional inclusion dependency mining systems and ILP approaches shows that MATILDA can scale to databases of various shapes and sizes that our competitors cannot handle. Overall, MATILDA mines about 50% more joinable patterns than competing systems across 35 benchmark databases while keeping runtimes competitive (sub-second to a few minutes depending on the database size). We also show that MATILDA finds not only all joinable patterns that the competitors find, but also a small portion of TGDs with more than one head atom, which none of our competitors can mine by design.

The rest of this paper is organized as follows: Section 2 reviews related work on pattern and rule mining, and Section 3 provides formal definitions. We then describe MATILDA in Section 4 and its recursive adaptation in Section 5. Section 6 reports our experimental results, and Section 7 concludes. All code and experimental results are publicly available at <https://github.com/fran-cois/MATILDA>.

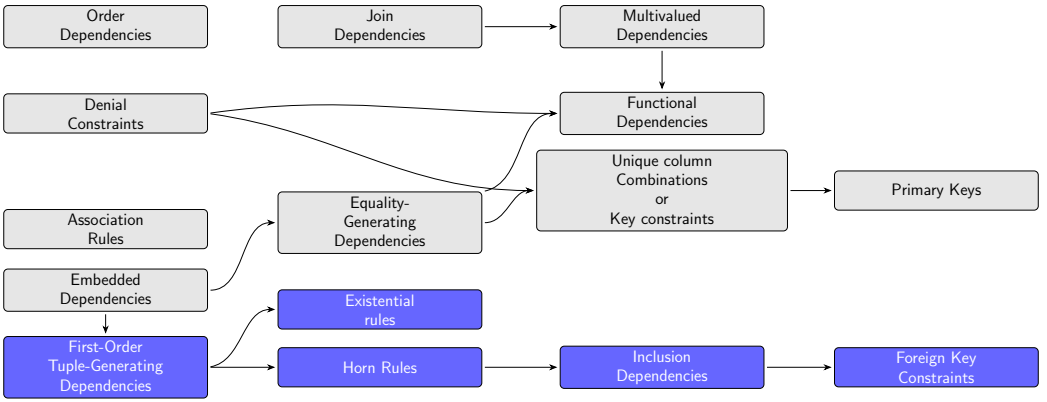


Fig. 2. Rule type taxonomy. An arrow means that the left type includes the right type. We mine the blue types.

## 2 Related work

Mining databases for patterns and dependencies has a long history. A wide variety of rule types have been studied (see, e.g., the surveys by Liu et al. [44] and Caruccio et al. [56] for overviews of functional, inclusion, conditional, and approximate dependencies). Figure 2 illustrates a taxonomy of rule types. An arrow from one type to another indicates that the former subsumes the latter. In the figure, we highlight in blue the classes of rules mined by our approach (MATILDA).

### 2.1 Mining Logical Patterns

**Association Rules** [16, 35, 78, 80, 81] are concerned with a set of sets of items (e.g., shopping baskets of individual shoppers). Association rules are then of the form “If a set contains these items, then it also contains those items”, with a famous example being that shoppers who buy diapers also usually buy beer.

**Denial constraints** (DCs) [27–29, 87, 99] are of the form  $\forall t_1, t_2, \dots, t_n : \neg(\phi(t_1, t_2, \dots, t_n))$  where  $\phi$  is a Boolean formula, with operators such as  $\leq, \geq, <, >, =, \neq$ , over the tuples  $t_1, t_2, \dots, t_n$ . The constraint states that no set of tuples in the database should make  $\phi$  true.

**Order dependencies** [6, 49, 85, 103] say that if one set of columns is ordered, so will be another one.

**Embedded dependencies** (ED) [5, 88, 111] (not to be confused with Embedded functional dependencies [107]) are constraints of the form  $\forall \vec{x}, \vec{y} : \phi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : \psi(\vec{x}, \vec{z}) \wedge \psi^=(\vec{x}, \vec{z})$  where  $\phi$

and  $\psi$  are conjunctions of logical atoms, and  $\psi^=$  is a conjunction of equality axioms. Embedded dependencies subsume a range of other dependencies.

**Equality-generating dependencies** (EGDs) [55, 88] are embedded dependencies where  $\psi$  is empty.

**Functional Dependencies** (FDs) [57, 95, 102] are the most prominent equality-generating dependencies, where  $\phi$  concerns only a single table. FDs say that if two rows in the table agree on their values for certain columns, then they also agree on certain other columns.

**Unique Column Combinations** (UCC) [12, 96], called **Key constraints** (KC) [24, 25, 62, 70, 97] in knowledge bases, are a subset of EGDs. A UCC is a set of columns in a database table that, when combined, uniquely identify each row. This means that no two rows can have the same values for the columns in the UCC. Primary keys (PKs) are UCCs that the database user defines.

**Tuple-generating dependencies** (TGDs) [86, 88] are embedded dependencies where  $\psi^=$  is empty. TGDs essentially state that some tuples in an instance imply the presence of some other tuples in that instance. In this work, we target this general class of dependencies, which includes subclasses such as guarded TGDs [9].

**Horn Rules** (HRs) [18, 21, 45, 93] are TGDs where  $\psi$  consists of a single atom. *Knowledge base rule mining* [22, 32, 45, 93, 100] and *Inductive logic programming (ILP)* [2, 3, 11, 20, 48, 51–54, 79, 90, 110] fall into this class.

**Inclusion Dependencies** (IDs) [4, 31, 104] are TGDs where  $\phi$  and  $\psi$  each consist of only a single atom. IDs say that the values of a subset of columns in one table are always present in another table. They are included in Horn rules, but some Horn rule miners do not mine all of them because IDs can link multiple columns, which not all Horn rule miners can find.

**Foreign Key Constraints** (FKC) [14, 47, 60, 112] are inclusion dependencies that are enforced in the database schema. Foreign key constraint mining consists in classifying what inclusion dependencies are foreign keys.

**TGDs include other dependencies** [40] that are neither Horn Rules nor inclusion dependencies. For example, TGDs include **existential rules**, i.e., rules that contain an existential quantifier in the rule head (as in the third example in the introduction).

**This paper** proposes MATILDA, an approach that mines TGDs on a database. It is known that there are currently no other approaches to this end [43]. One may think that TGDs can be simulated by Horn rules: The existential rule  $\forall \vec{x}, \vec{y} : \phi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : p(\vec{x}, \vec{z}) \wedge q(\vec{x}, \vec{z})$  can be split into the Horn rules  $\forall \vec{x}, \vec{y} : \phi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : p(\vec{x}, \vec{z})$  and  $\forall \vec{x}, \vec{y}, \vec{z} : \phi(\vec{x}, \vec{y}) \wedge p(\vec{x}, \vec{z}) \Rightarrow q(\vec{x}, \vec{z})$ . However, (1) finding these combinations still requires a combinatorial search and (2) the confidence of the combined rule will still have to be computed. Our approach identifies all TGDs and their associated confidence values in a single iteration – faster or comparable in performance to Horn rule mining alone.

## 2.2 Mining techniques

All approaches that mine TGDs or subsets thereof (like AMIE [45], Popper [10], SPIDER [38], and our MATILDA) face a large search space. The methods differ in how they traverse this space.

AMIE uses a rule expansion tree to generate candidate rules, where nodes correspond to Horn rule prefixes, and edges represent valid rule extensions (i.e., adding one atom to the body or head while maintaining variable connectivity). This method is tailored to Horn rules with a single atom in the head. Popper enumerates candidate hypotheses via a declarative bias. This technique cannot take into account joinability between columns, which leads to longer runtimes. SPIDER can mine only unary inclusion dependencies (INDs), and hence, candidates are simply all column pairs with non-zero overlap. MATILDA, in contrast, constructs a constraint graph, where each node represents a pair of columns. This allows the efficient enumeration of rules that go beyond Horn rules and inclusion dependencies.

For pruning, AMIE uses a combination of syntactic and count-based methods. In Popper, each failed hypothesis generates constraints that prune its generalizations (if too specific) or specializations (if too general). SPIDER uses the ordering of distinct values to detect non-inclusions efficiently. MATILDA prunes by the metric of support and by exploiting the dualism between a TGD and its inverse, where head and body are swapped.

To evaluate a rule, AMIE computes support and confidence values, using the Partial Completeness Assumption (PCA) to deal with the Open World setting. In Popper, in contrast, evaluation is binary: a hypothesis is either consistent with all positive and negative examples or not. SPIDER can mine inclusion dependencies even when they hold only for a small fraction of the data (i.e., approximate INDs with low support). MATILDA generalizes AMIE's notions of confidence and support from binary facts to  $n$ -ary tables, and can thus mine approximate patterns on non-binary tables.

### 2.3 Other Approaches

A **schema mapping** [34, 61, 77] establishes relationships between a source and a target database schema. Given attribute correspondences between schemas, *source-to-target TGDs* describe instance transformations from the source database to the target one. In data exchange terminology, we mine a superset of what is usually called *target TGDs*, i.e., TGDs on one database schema.

**Statistical schema induction** [50] derives schema-level information from RDF data using statistical methods like association rule mining. The process involves querying RDF repositories to identify classes and properties, constructing transaction tables for resources, and mining these tables for association rules, while we are concerned with mining general TGDs on databases.

**TGD decidability** [36, 74] is concerned with finding whether it can be decided that TGDs of a certain class hold or do not hold.

**Ontological reasoning** [26, 39, 55, 58] uses TGDs to infer new knowledge or make logical deductions in an ontology. It is thus an application of the TGDs that our approach mines.

**Causal Inference** [15, 23, 75] aims to establish causal relationships between variables with a graph structure. Causal graph inference differs from rule mining, focusing on understanding cause-and-effect relationships rather than simply identifying patterns.

**Knowledge embeddings** [8, 33, 66, 91, 98, 106, 108] represent structured knowledge as vectors in a high-dimensional space with the goal of capturing semantic relationships between entities.

**Rule mining with knowledge embeddings** [42, 69] uses embeddings to improve the pruning of rule mining. We do not use such methods as we consider databases without prior knowledge.

**Join dependencies (JDs)** [19, 65] are constraints that originate in database-normalization theory. A JD stipulates that a relation  $R$  can be losslessly decomposed into smaller relations  $R_1, \dots, R_n$  such that  $R = R_1 \bowtie \dots \bowtie R_n$ . In other words, whenever tuples drawn from each  $R_i$  agree on their shared attributes, their natural join must also be present in  $R$ . Join dependencies strictly generalize **multivalued dependencies (MVDs)**, which arise when  $n = 2$ . Despite capturing valuable structural regularities, JDs and MVDs are rarely mined in practice because of their combinatorial complexity, so they remain largely of theoretical interest. We mention them here for completeness.

**Join Discovery** systems and algorithms [30, 82, 84, 89, 101, 105] compute join paths across relations and schemas automatically. Such joins are an input to our method, as we aim to discover patterns in such joinable data.

## 3 Definitions

### 3.1 Preliminaries

We start with some standard definitions.

*Definition 3.1 (Relational Database).* A relational database (database for short) is a set of named tables (also called relations) [1, 88]. We assume that each table has a unique name. A table  $T$  consists of an ordered set of attribute names (attributes for short)  $A(T) = \{a_1, \dots, a_n\}$ , their associated domains  $\{D_1, \dots, D_n\}$  (each of which is a set of values), and the tuples (or rows), each of which is an element of  $D_1 \times D_2 \times \dots \times D_n$ . The arity of  $T$  is  $n$ . Without loss of generality, we assume that all attribute names (across all tables) are distinct. We write  $T(v_1, \dots, v_n)$  to say that a tuple  $\langle v_1, \dots, v_n \rangle$  is in  $T$ . We then say that  $T(v_1, \dots, v_n)$  holds. This notion generalizes to first-order formulas. We write  $a_i(\langle v_1, \dots, v_n \rangle) = v_i$  to select a value from a tuple. We write  $T(a_i = v_i, \dots, a_j = v_j)$  for attribute names  $a_i, \dots, a_j \in A(T)$  and values  $v_i \in D_i, \dots, v_j \in D_j$  to say that there exists at least one tuple  $t$  in  $T$  with  $a_i(t) = v_i, \dots, a_j(t) = v_j$ . We write  $T.a_i = \{v_i \mid \exists v_1, \dots, v_n : T(v_1, \dots, v_n)\}$  to select all values of a given attribute from the table. A *key*  $K$  of a table  $T$  is a minimal set of attributes so that no distinct tuples share the values of these attributes:  $\forall v, w : T(v) \wedge T(w) \wedge (\forall a \in K : a(v) = a(w)) \Rightarrow v = w$ , and no proper subset of  $K$  has this property. Let  $S$  and  $T$  be two tables and let  $K \subseteq A(S)$  be a key of  $S$ . A *foreign key* of  $T$  to  $S$  is a set  $F \subseteq A(T)$  and an injective function  $f : F \rightarrow K$  such that  $\forall v : T(v) \Rightarrow \exists w : S(w) \wedge \forall b \in F : f(b)(w) = b(v)$ . For our purposes, the *schema* of a database is the set of tables with their attributes, domains, key constraints, and foreign key constraints, but without their tuples.

**EXAMPLE 1.** The Lineage table from Figure 1 has the attributes “parent” and “child”, i.e.,  $A(\text{Lineage}) = \{\text{parent}, \text{child}\}$ . Both have the domain “person names”. The arity of the table is 2. We have  $\text{Lineage}(\text{Elvis}, \text{Lisa})$ ,  $\text{parent}(\langle \text{Elvis}, \text{Lisa} \rangle) = \text{Elvis}$ , and  $\text{Lineage.parent} = \{\text{Elvis}, \text{Priscilla}, \text{Lisa}, \text{Danny}\}$ , and we know that there exists a tuple with Lisa as a parent,  $\text{Lineage}(\text{parent} = \text{Lisa})$ . The set  $\{\text{parent}\}$  is a key of Lineage, while the set  $\{\text{child}\}$  is not, as the same child appears in different rows. The function  $\{\text{Lineage.parent}\} \rightarrow \{\text{Residence.person}\}$  is a foreign key.

In what follows, we assume a fixed database and schema.

*Definition 3.2 (Tuple-Generating Dependency).* First-Order Tuple-Generating Dependencies (TGDs) are first-order logic sentences of the form

$$\forall \vec{x}, \vec{y} : \phi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : \psi(\vec{x}, \vec{z})$$

Here,  $\vec{x}, \vec{y}, \vec{z}$  are sets of variables,  $\phi$  and  $\psi$  are conjunctions of atoms (called the body and the head, respectively), and each atom is of the form  $t(x_1, \dots, x_n)$ , where  $t$  is a table name with  $n$  attributes, and the  $x_i$  come from the indicated set(s) of variables.  $\psi$  cannot be empty.

**EXAMPLE 2.** “Every parent has a residence” is a TGD of the form  $\forall p, c : \text{Lineage}(p, c) \Rightarrow \exists l, s : \text{Residence}(p, l, s)$ . Here,  $\vec{x} = \{p\}$ ,  $\vec{y} = \{c\}$ , and  $\vec{z} = \{l, s\}$ . This TGD can be written in a simpler form as  $\forall x : \text{Lineage}(\text{parent} = x) \Rightarrow \text{Residence}(\text{person} = x)$ .

A TGD is *recursive* if it contains the same table more than once. A TGD is *transitively connected* if any two atoms in the TGD are transitively connected through shared variables. TGDs that do not have this property link disconnected conditions and are typically less meaningful (consider, e.g.,  $\forall p, c : \text{Lineage}(p, c) \Rightarrow \exists j, l, z : \text{Residence}(j, l, z)$ ). This is why prior work, as well as our own, focuses exclusively on connected patterns [21, 45, 93]. Two TGDs are *logically equivalent* if they hold on exactly the same databases. For example, swapping two atoms in the body of a TGD results in a logically equivalent TGD. One typically aims to mine only one TGD for each equivalence class of TGDs.

### 3.2 Metrics

TGDs make predictions, as follows:

*Definition 3.3 (Prediction).* The *Prediction* of a TGD  $R = \forall \vec{x}, \vec{y} : \phi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : \psi(\vec{x}, \vec{z})$  is the set of tuples  $\vec{x}$  where  $R$  holds:

$$\text{Prediction}(R) = \{\vec{x} \mid \exists \vec{y}, \vec{z} : \phi(\vec{x}, \vec{y}) \wedge \psi(\vec{x}, \vec{z})\}$$

As in AMIE, a prediction is the head atom obtained from any instantiation whose body atoms all occur in the database, with the same bindings for  $\vec{x}$ .

We can now use the notion of predictions to define support and confidence of approximate TGDs:

*Definition 3.4 (Support).* The *support* of a TGD  $R = \forall \vec{x}, \vec{y} : \phi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : \psi(\vec{x}, \vec{z})$  is the ratio of tuples where  $R$  holds to the total number of tuples  $\vec{x}$  for which all the head atoms of  $R$  hold:

$$\text{Support}(R) = \frac{|\text{Prediction}(R)|}{|\{\vec{x} \mid \exists \vec{z} : \psi(\vec{x}, \vec{z})\}|}$$

We thus use a relative definition of support, which is called *head coverage* in AMIE [45].

**EXAMPLE 3.** Consider the rule “All children are married”, formalized as  $\forall p, c : \text{Lineage}(p, c) \Rightarrow \exists s : \text{Marriage}(c, s)$ . With the data in Figure 1, the set of married people is {Elvis, Priscilla, Lisa, Danny} (size 4), while the set of married children is {Lisa} (size 1). Hence the support (head coverage) is  $\frac{\#\{c \mid \exists p, s : \text{Lineage}(p, c) \wedge \text{Marriage}(c, s)\}}{\#\{x \mid \exists s : \text{Marriage}(x, s)\}} = \frac{1}{4} = 0.25$ .

*Definition 3.5 (Confidence).* The *confidence* of a TGD  $R = \forall \vec{x}, \vec{y} : \phi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : \psi(\vec{x}, \vec{z})$  is the ratio of tuples  $\vec{x}$  for which  $R$  holds to the total number of tuples  $\vec{x}$  for which all the body atoms hold:

$$\text{Confidence}(R) = \frac{|\text{Prediction}(R)|}{|\{\vec{x} \mid \exists \vec{y} : \phi(\vec{x}, \vec{y})\}|} \quad \left| \quad \text{Confidence}(R) = \begin{cases} 1, & \text{if } \exists \vec{x}, \vec{z} : \psi(\vec{x}, \vec{z}), \\ 0, & \text{otherwise.} \end{cases} \right.$$

if  $\phi$  is empty

**EXAMPLE 4.** For the same rule  $\forall p, c : \text{Lineage}(p, c) \Rightarrow \exists s : \text{Marriage}(c, s)$ , the confidence (with  $\vec{x} = (c)$ ) is computed over body tuples: there are 2 children LINEAGE, and exactly 1 of them is married (Lisa). Thus  $\frac{\#\{(c) \mid \exists s : \text{Lineage}(p, c) \wedge \text{Marriage}(c, s)\}}{\#\{(c) \mid \text{Lineage}(p, c)\}} = \frac{1}{2} = 0.5$ .

Note that, as we work with databases, confidence is computed under the Closed World Assumption, meaning that any tuple that is predicted but not in the database reduces the confidence. One may think that the confidence of a TGD  $\forall \vec{x}, \vec{y} : \phi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : p(\vec{x}, \vec{z}) \wedge q(\vec{x}, \vec{z})$  can be computed from the confidences of the Horn rules  $\forall \vec{x}, \vec{y} : \phi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : p(\vec{x}, \vec{z})$  and  $\forall \vec{x}, \vec{y}, \vec{z} : \phi(\vec{x}, \vec{y}) \wedge p(\vec{x}, \vec{z}) \Rightarrow q(\vec{x}, \vec{z})$ , for example by multiplying the confidences. This is, however, not the case, as a simple example shows. Consider the TGD “Married partners must share some residence”,  $\forall x, y : \text{Marriage}(x, y) \Rightarrow \exists l, s : \text{Residence}(x, l, s) \wedge \text{Residence}(y, l, s)$  with a confidence of  $\frac{2}{4} = 0.5$ . Now, let us look at the two constituent Horn rules:  $\forall x, y : \text{Marriage}(x, y) \Rightarrow \exists l, s : \text{Residence}(x, l, s)$  (“if  $x$  is married then has a residence”) and  $\forall x, y, l, s : \text{Marriage}(x, y) \wedge \text{Residence}(x, l, s) \Rightarrow \text{Residence}(y, l, s)$  (“if  $x$  is married with  $y$  and lives in  $l$  then  $y$  lives in  $l$ ”). Their confidences are  $\frac{3}{4}$  and  $\frac{3}{3}$ . Now,  $\frac{3}{4} \times \frac{3}{3} = \frac{3}{4} \neq 0.5$ .

Thus, the confidence of a TGD with more than one atom in the head cannot be computed by a multiplication of its constituent Horn Rules. However, we have:

**PROPOSITION 3.6.** For any Tuple-Generating Dependency (TGD)  $R = \forall \vec{x}, \vec{y} : \phi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : \psi(\vec{x}, \vec{z})$  and its symmetric counterpart  $R^* = \forall \vec{x}, \vec{y} : \psi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : \phi(\vec{x}, \vec{z})$ , we have  $\text{Support}(R) = \text{Confidence}(R^*)$  and  $\text{Confidence}(R) = \text{Support}(R^*)$ .

PROOF. By definition, we have

$$\text{Confidence}(R) = \frac{|\text{Prediction}(R)|}{|\{\vec{x} \mid \exists \vec{y} : \phi(\vec{x}, \vec{y})\}|}, \quad \text{Support}(R) = \frac{|\text{Prediction}(R)|}{|\{\vec{x} \mid \exists \vec{y} : \phi(\vec{x}, \vec{y})\}|}$$

Since the predictions are computed by the conjunction of  $\psi$  and  $\phi$ , which is the same for  $R$  and  $R'$ , we have

$$\text{Prediction}(R) = \text{Prediction}(R').$$

The claim  $\text{Support}(R) = \text{Confidence}(R')$  follows analogously.  $\square$

This proposition halves the amount of computation by calculating each metric just once and then applying it to its counterpart.

### 3.3 Task Definition

*Definition 3.7 (TGD mining).* Given a database, TGD mining is the task of finding all transitively connected TGDs (up to logical equivalence), with their confidence and support.

EXAMPLE 5. In our toy database from Figure 1, the goal could be to mine “If someone is a parent then the person is married and has a residence”:

$$\forall x_1, y_1 : \text{Lineage}(x_1, y_1) \Rightarrow \exists z_1, z_2, z_3 : \text{Residence}(x_1, z_1, z_2) \wedge \text{Marriage}(x_1, z_3)$$

The first challenge in mining TGDs is that there are infinitely many logically equivalent variations of the same TGD: variables can be renamed, and atoms can be swapped and duplicated. Thus, one has to implement extensive duplication checks, as in rule mining algorithms [45]. We counter this challenge by introducing a redundancy-free graph representation of TGDs. The second challenge is that the search space is very large. To see this, let us assume that there are  $n$  attributes in total in our database ( $n = 7$  in our toy example). To mine all possible TGDs, consider all possible equalities between the attributes, such as equality  $\text{Lineage.parent} = \text{Residence.person}$  in our example. Equalities generate equivalence classes of attributes, such as  $\{\text{Lineage.parent} = \text{Residence.person} = \text{Marriage.partner1}\}$ , and each equivalence class corresponds to one variable in the TGD. To enumerate all possible TGDs, one thus needs to enumerate all possible partitions of the set of attributes. The number of possible partitions is the Bell number of  $n$  [113]:  $B_n = \sum_{k=0}^{n-1} \binom{n}{k} B_k$ , with the base case  $B_0 = 1$ .

EXAMPLE 6. For a database with 75 attributes (i.e.,  $n = 75$ ), the number of required permutations,  $B_{75}$ , exceeds  $10^{80}$ , surpassing the estimated number of hydrogen atoms in the observable universe.

## 4 MATILDA

We can now present our method, MATILDA. We first discuss the method for the case of non-recursive rules, and then extend it to recursive rules in Section 5.

### 4.1 Constraint Graph

One may think that the simplest way to represent the space of possible TGDs would be a graph where the nodes are attributes and the edges say whether two attributes are joinable. However, such a graph would allow enumerating attributes and not TGDs. Therefore, we construct a different graph, which we call the *constraint graph* (Figure 3 shows the constraint graph for our toy example). The intuition of the constraint graph is that each node represents a constraint that can be added to a TGD. For example, consider the TGD that says that parents have a residence:  $\forall p, c : \text{Lineage}(p, c) \Rightarrow \exists l, z : \text{Residence}(p, l, z)$ . Adding the constraint that the parent is married,  $\text{Lineage.parent} = \text{Marriage.partner1}$ , yields the TGD  $\forall p, c : \text{Lineage}(p, c) \Rightarrow \exists l, z, s : \text{Residence}(p, l, z) \wedge \text{Marriage}(p, s)$ . In this way, traversing the graph corresponds to adding more constraints to our TGD.

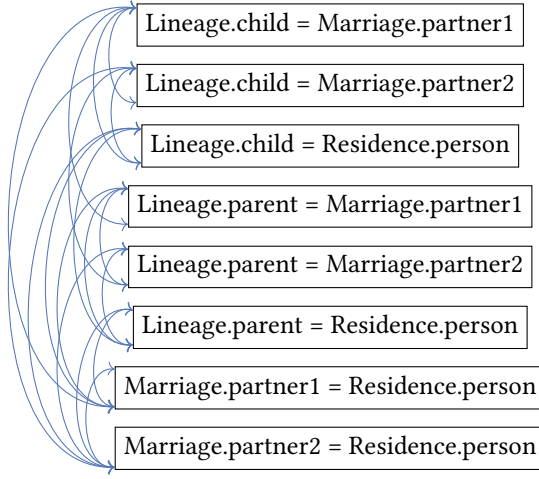


Fig. 3. Constraint graph of our example database, with the tables Lineage, Residence, and Marriage. Each node is table-linked with each other node. The arrows show nodes that are perfectly table-linked (see Definition 4.2).

Let us now make this intuition more formal: We assume a complete order of the tables of the database and a complete order of the attributes of each table (e.g., the lexicographical order). We also assume a joinability relation between attributes:

*Definition 4.1 (Joinability Relation).* A joinability relation is a symmetric and reflexive binary relationship between the columns of a database schema that indicates when two columns are semantically related, i.e., when they can be meaningfully equated in a join predicate.

EXAMPLE 7. In our toy database, *Lineage.parent* and *Residence.person* are joinable.

The joinability relation can be specified manually by users or mined using schema-matching algorithms [73]. In particular, foreign keys are natural joinability relations. The joinability relation is not transitive: Given a table  $T$  with three attributes  $T.a$ ,  $T.b$ , and  $T.c$ , one can have that  $T.a$  and  $T.b$  are joinable (which commonly implies  $T.a \cap T.b \neq \emptyset$ ) and that  $T.b$  and  $T.c$  are joinable ( $T.b \cap T.c \neq \emptyset$ ), but this does not mean that  $T.a$  and  $T.c$  are joinable ( $\not\Rightarrow T.a \cap T.c \neq \emptyset$ ).

The notion of joinability allows us to define table-linked attributes:

*Definition 4.2 (Table-linked Attributes).* Two attributes  $T.a$  and  $T'.a'$  are table-linked if they share the same table, i.e., if  $T = T'$ .

Two pairs of joinable attributes,  $v_1 = (T_1.a_1, T'_1.a'_1)$ ,  $v_2 = (T_2.a_2, T'_2.a'_2)$  are table-linked if at least one attribute from  $v_1$  is table-linked with at least one attribute from  $v_2$ . If  $v_1$  and  $v_2$  share an attribute, then  $v_1$  and  $v_2$  are *perfectly table-linked*.

*Definition 4.3 (Constraint Graph).* The constraint graph of a database is the directed unlabeled acyclic graph where

- A node  $(T.a, T'.a')$  is a pair of joinable attributes such that  $T.a < T'.a'$ . We write the node as  $T.a = T'.a'^2$ .
- An edge  $(v, v')$  links two table-linked nodes  $v$  and  $v'$  with  $v < v'$ .

<sup>2</sup> $T.a < T'.a'$  refers to the lexicographic ordering. Most notably, the two attributes cannot be equal.  $T.a = T'.a'$  is the way we write the node  $(T.a, T'.a')$ , because the node joins two attributes; see Figure 3.

Constraint graphs will help us to find proto-rules, i.e., patterns that give rise to TGDs.

*Definition 4.4 (Proto-Rule).* Given a path  $P$  in a constraint graph, a proto-rule is the conjunction of atoms obtained as follows:

- (1) Create a conjunction with one atom for each table mentioned in  $P$ , each with fresh variables.
- (2) Construct the replacement relation  $\rightarrow_R$ , which contains  $T.a \rightarrow_R T'.a'$  iff  $T.a = T'.a'$  is in  $P$ .
- (3) For each attribute  $T.a$ , replace its variable by the variable of the attribute  $\max\{T'.a' \mid T.a \xrightarrow*_R T'.a'\}$ .

*Definition 4.5 (Proto-Rule Length).* For a path  $P$  in the constraint graph, the proto-rule length is the length of  $P$ .

EXAMPLE 8. Consider the graph in Figure 3, and consider the following path in that constraint graph, which contains two nodes:

$$\text{Lineage.parent} = \text{Marriage.partner1}$$

$$\text{Lineage.parent} = \text{Residence.person}$$

We obtain the following conjunction:

$$\text{Lineage}(p, c) \wedge \text{Marriage}(p_1, p_2) \wedge \text{Residence}(p_3, l, s)$$

We obtain the following replacement relation:

$$\text{Lineage.parent} \rightarrow_R \text{Marriage.partner1} \rightarrow_R \text{Residence.person}$$

This means that all the variables that are concerned by this replacement relation have to be replaced by the variable of *Residence.person*, i.e. by  $p_3$ :

$$\text{Lineage}(p_3, c) \wedge \text{Marriage}(p_3, p_2) \wedge \text{Residence}(p_3, l, s)$$

*Definition 4.6 (Proto-Rule Count).* The count of a proto-rule is the number of possible bindings of the variables so that the conjunction of the proto-rule holds on the database.

EXAMPLE 9. The count of the proto-rule in Example 8 is the number of elements in the cross product of *Lineage*, *Residence*, and *Marriage* on which the proto-rule constraint holds – which is three.

Proto-rules become TGDs by splitting them into body and head:

*Definition 4.7 (Split).* A split of a proto-rule is a partition of the set of tables of the proto-rule into two sets, the head and the body. A split proto-rule is a pair of a proto-rule and one of its splits  $\langle B, H \rangle$ . A split proto-rule can be written as a TGD:

$$\forall \vec{x}, \vec{y} : \phi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : \psi(\vec{x}, \vec{z})$$

where  $\phi, \psi$  are a partition of the atoms of the proto-rule so that all atoms with table names of  $B$  appear in  $\phi$ . Each variable  $\alpha$  of the proto-rule is assigned as follows: If  $\alpha$  belongs to at least one table in the body and to one table in the head, then  $\alpha \in \vec{x}$ . If  $\alpha$  belongs only to body tables, then  $\alpha \in \vec{y}$ , otherwise  $\alpha \in \vec{z}$ .

EXAMPLE 10. Consider again the proto-rule of Example 5. By splitting this proto-rule into the body  $\{\text{Lineage}\}$  and the head  $\{\text{Residence}, \text{Marriage}\}$ , we obtain:

$$\forall x_1, y_1 : \text{Lineage}(x_1, y_1) \Rightarrow \exists z_1, z_2, z_3 : \\ \text{Residence}(x_1, z_1, z_2) \wedge \text{Marriage}(x_1, z_3)$$

PROPOSITION 4.8. For any non-recursive TGD that joins only joinable attributes, the constraint graph contains a proto-rule that gives rise to that TGD.

PROOF. Consider a target TGD of the form  $\forall \vec{x}, \vec{y} : \psi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : \phi(\vec{x}, \vec{z})$ . We apply the reverse of the instantiation process of Definition 4.7: We first construct the split into head and body by partitioning the set of tables that appear in the TGD – which is possible because each table name appears at most once in the TGD. We can then reconstruct the nodes of the constraint graph: If two atoms  $t(x_1, \dots, x_n)$  and  $t'(x'_1, \dots, x'_n)$  of the target TGD share a common variable  $x_i = x'_i$ , we derive the constraint graph node  $t.i = t'.i'$ . This constraint (or  $t'.i' = t.i$ ) must appear in the constraint graph because the graph contains all pairs of joinable attributes, and the TGD connects only joinable attributes. Furthermore, if two pairs of constraints share an attribute (i.e., constraints of the form  $t.i = t'.i'$  and  $t.i = t''.i''$ ), these constraint pairs are linked in the constraint graph by construction. Since we consider only connected TGDs, each atom of the TGD is linked to at least one other atom in this manner, ensuring that all atoms are transitively connected within the graph. As a result, the subgraph formed by these linked constraints is itself connected and corresponds to a proto-rule, which is the conjunction of the TGD's atoms.  $\square$

This entails that mining all split proto-rules allows mining all non-recursive TGDs up to equivalence.

## 4.2 Algorithm

The heart of MATILDA is Algorithm 1<sup>3</sup>. It takes a constraint graph and the maximum TGD length (the maximum number of constraints in the proto-rule) as input. It then outputs a continuous sequence of proto-rules. Initially, the algorithm iterates over all nodes in the graph, initiating a depth-first search (DFS) from each node. The current starting node is then added to the set of visited nodes, marking it as visited. This node is also appended to the proto-rule. If the proto-rule is smaller than the maximal length, the algorithm calls itself with the new proto-rule and each neighbor of the current node. We exclude neighbors that have been visited. Upon returning from the recursive call, the neighbor is removed from the visited set and from the proto-rule list. The algorithm backtracks and explores alternative paths. This approach systematically explores all paths in the graph and filters out proto-rules that are over the maximal length or that have a proto-rule count of zero.

PROPOSITION 4.9. *Given a database, Algorithm 1 generates all proto-rules that have a count greater than zero and a length smaller than or equal to the given `max_length`.*

PROOF. Given a proto-rule  $R$  that has a count greater than zero, and a length smaller than or equal to `max_length`, we show how Algorithm 1 finds this rule. Consider the lexicographically smallest node of  $R$ . Since Algorithm 1 iterates through all nodes of the constraint graph in Line 9, it will eventually consider this node. The algorithm then explores all neighbors of this node recursively. Since  $P$  is a path in the graph, the algorithm will also explore  $P$  recursively. Since  $P$  has a length smaller than or equal to `max_length`, all prefixes of  $P$  will also have a length smaller than or equal to `max_length`. Since  $P$  has a count greater than zero, all prefixes of  $P$  will also fulfill this condition.  $\square$

PROPOSITION 4.10. *Given a database, Algorithm 1 generates proto-rules without duplicates.*

PROOF. Given a constraint graph  $CG$ , let us assume that Algorithm 1 generates the same proto-rule twice, i.e., it generates two equivalent proto-rules  $PR_1$  and  $PR_2$ . We first note that Algorithm 1 visits each path in the constraint graph exactly once: each node is added at most once per branch thanks to the visited set, and edges are followed only in the forward direction of the current path.

<sup>3</sup>We show our algorithms in Python instead of pseudo-code to avoid ambiguity, help reproducibility, and ensure correspondence between our system and the paper.

Algorithm 1. Generate Proto-rules

```

1 def dfs(
2     graph: ConstraintGraph,
3     max_length: int = 3,
4     start_node: JoinableAttributes = None,
5     visited: set[JoinableAttributes] = set(),
6     proto_rule: ProtoRule = [],
7 ) -> Iterator[ProtoRule]:
8     if start_node is None:
9         for next_node in graph.nodes:
10            yield from dfs(graph, max_length, next_node, visited, proto_rule)
11        return
12    if len(proto_rule) >= max_length: return
13    visited.add(start_node)
14    proto_rule.append(start_node)
15    yield proto_rule
16    for next_node in graph.neighbors(start_node) - visited:
17        if count(proto_rule + next_node) > 0:
18            yield from dfs(graph, max_length, next_node, visited, proto_rule)
19    visited.remove(start_node)
20    proto_rule.pop()

```

Consequently, two different branches cannot traverse the same sequence of nodes in the same order; no duplicate proto-rule is emitted.  $\square$

Algorithm 2. Generate Splits

```

1 def split_proto_rule(proto_rule: ProtoRule):
2     tables = extract_tables(proto_rule)
3     return [ (body, tables - body) for body in powerset(tables)
4             if len(body) <= len(tables) / 2 and len(body) > 0 ]

```

Each proto-rule is then split into body and head (Definition 4.7) by Algorithm 2. This algorithm takes as input a proto-rule and outputs a set of splits. The algorithm starts by extracting the tables involved in the proto-rule (extract\_tables function) and then iterates over all possible subsets of the tables (powerset function). Each subset becomes a body, and its complement becomes a head. We do not return splits where the body is empty. We also do not return splits where the body has more atoms than the head. This is because our metrics are symmetric (Proposition 3.6).

*Support and confidence thresholds.* MATILDA uses two user-defined thresholds, *support* and *confidence*, to filter candidate TGDs during enumeration. The *support* threshold discards proto-rules whose underlying join graph occurs fewer times than the specified value; the *confidence* threshold removes candidate TGDs whose conditional probability falls below the threshold. We use the support threshold to prune early, before splits, and the confidence threshold to filter the rules in the output, after splits. Lower thresholds increase recall but blow up the search space; higher thresholds speed up execution at the risk of missing low-support or low-confidence TGDs. A common practice is to start with moderate settings (e.g., 5% support and 0.8 confidence) and relax them if the output is too sparse.

We use these algorithms in our main algorithm, Algorithm 3. This algorithm takes as input the maximal TGD length threshold and a mapping between joinable attributes. This mapping contains, for each attribute, a set of attributes with which it can be joined (symmetrically). The algorithm outputs pairs of TGDs with their metrics (support and confidence). This works by first creating the constraint graph (Definition 4.3). Then, the algorithm runs through all proto-rules and through all splits of the proto-rule. For each split proto-rule, we compute the confidence and support (Definitions 3.4 and 3.5). Since our metrics are symmetric (Proposition 3.6), we can generate two rules for each split, as per Definition 4.7.

Algorithm 3. MATILDA

```

1 def MATILDA(
2     joinable:Dict[Attribute, [Attribute]],
3     max_length : int = 3
4 ):
5     graph = create_constraint_graph(joinable)
6     for proto_rule in dfs(graph,max_length):
7         if count(proto_rule) > 0:
8             for split in split_proto_rule(proto_rule):
9                 metrics = comp_metrics(proto_rule, split)
10                if metrics.support > 0:
11                    yield (createTGD(proto_rule, split), metrics)
12                if metrics.swap().support > 0:
13                    yield (createTGD(proto_rule, split.swap()), metrics.swap())

```

### 4.3 Theoretical Analysis

*Correctness.* From Proposition 4.8, Proposition 4.9, and the fact that Algorithm 1 generates all proto-rules that have a count greater than zero and a number of constraints less than  $max\_length$  we directly have:

**PROPOSITION 4.11.** *Algorithm 3 generates all non-recursive TGDs that hold in the given database and that have a proto-rule with a non-zero count and a number of constraints smaller than or equal to  $max\_length$ , without generating duplicate TGDs.*

**PROOF.** We show that MATILDA (Algorithms 1–3) enumerates every admissible non-recursive TGD exactly once.

*Completeness.* Fix a TGD  $R$  that has a proto-rule with a non-zero count and a number of constraints smaller than or equal to  $max\_length$ . Proposition 4.8 yields a path  $P$  in the constraint graph whose proto-rule reduces to  $R$ . Since  $count(P) > 0$  and  $|P| \leq max\_length$ , Algorithm 1 outputs  $P$  (Proposition 4.9). Algorithm 2 then considers every admissible body-head split of the tables of  $P$ ; among them is a split  $S$  that matches  $(body(R), head(R))$  up to symmetry. When Algorithm 3 processes  $\langle P, S \rangle$ , the support test in line 10 succeeds (since  $R$  holds in the data), so lines 11 and 12 emit  $R$ . Thus  $R$  is produced.

*Soundness.* Any rule emitted by Algorithm 3 (i) originates from a proto-rule produced by Algorithm 1, and hence obeys the length and positive-count constraints, and (ii) passes the support check in line 10. Therefore, every output rule satisfies the required properties.

*Uniqueness.* Assume two identical TGDs were emitted. Then these cannot have distinct proto-rules as Algorithm 1 outputs each proto-rule once (Proposition 4.10). The TGDs cannot have different splits either, because Algorithm 2 keeps only one of  $\langle B, H \rangle$  and  $\langle H, B \rangle$  ( $|B| \leq |H|$ ), and Algorithm 3 outputs at most one orientation per split. Hence, no duplicates can occur.  $\square$

*Complexity.* To determine the complexity of Algorithm 3, we focus on the number of *split proto-rules* generated based on the number of joinable attributes  $n$ . We decompose the complexity into:

- (1) the number of start nodes,
- (2) the number of proto-rules generated from each start node,
- (3) and the number of possible splits for each proto-rule.

There are  $n$  start nodes (one per pair of joinable attributes). In the worst case, every pair of joinable attributes is joinable with every other one, making the constraint graph a complete acyclic orientation with  $\binom{n}{2}$  edges. From a single start node, the number of proto-rules (simple paths) that can be generated is  $2^{n-1} - 1$ : Any subset of the remaining  $n - 1$  nodes, taken in increasing order, yields exactly one path. Summing over all starts does not multiply this by  $n$  (each path has a unique smallest node), so the total number of distinct proto-rules across all starts is  $2^n - n - 1$ .

For a proto-rule  $r$ , let  $T(r)$  be the number of tables used by  $r$ . In the worst case,  $T(r)$  can be as large as  $n$ , and each proto-rule can be split in up to  $2^{T(r)}$  ways; thus we take the upper bound  $2^n$  per rule. Multiplying these factors gives the following worst-case complexity:

$$O((2^n - n - 1) \times 2^n) = O(2^{2n}).$$

For comparison, the naive approach relies on generating all partitions of  $n$  attributes (which has Bell-number complexity) and then enumerating all splits (up to  $2^n$ ). In practice, the naive approach quickly becomes infeasible: Already for  $n = 13$ , it would require at least  $3 \times 4,213,597$  SQL queries (one query for each of prediction, coverage, and support). With 0.01s per query, it would not finish in a day. By contrast, our method can prune on two fronts:

- (1) it considers only *joinable* attributes, and
- (2) it discards proto-rules with insufficient count.

These two pruning strategies allow our approach to run in practice.

Completeness: (i) Algorithm 1 generates all admissible proto-rules (Proposition 4.9); (ii) Algorithm 2 iterates over all legal splits of those proto-rules; (iii) Algorithm 3 outputs both directions of every split. Consequently, no valid TGD is missed.

## 5 MATILDA with recursive TGDs

### 5.1 Algorithm

In this section, we adapt MATILDA to mine recursive TGDs, i.e., TGDs that contain the same table more than once. Mining recursive TGDs requires adapting the definitions of attribute, proto-rule, split proto-rule, and TGD metrics. We start with the definition of qualified tables and attributes:

*Definition 5.1 (Qualified Table and Attribute).* A *qualified table* is a table with a natural number called the table occurrence  $i$ , written  $t_i$ . A *qualified attribute* is a triple of a table  $t$ , a table occurrence  $i$ , and one of its attributes  $a$ , written  $t_i.a$ . Lexical order defines a total order on qualified tables and on qualified attributes.

In what follows, we write TGDs with qualified tables and attributes, so that each atom with the same table name as another atom has a different table occurrence.

EXAMPLE 11. A recursive TGD is  $\text{Marriage}_0(x, y) \Rightarrow \text{Marriage}_1(y, x)$ .

*Definition 5.2 (N-Recursive Constraint Graph).* An  $N$ -Recursive Constraint Graph for a natural number  $N$  is a directed unlabeled acyclic graph (similar to Definition 4.3), where

- A node  $(T_i.a, T'_j.a')$  is a pair of joinable qualified attributes such that  $T_i.a < T'_j.a'$  and  $i, j \leq N$
- An edge  $(v, v')$  links two table-linked nodes  $v$  and  $v'$  with  $v < v'$ .

The recursive graph can then generate proto-rules:

*Definition 5.3 (Recursive Proto-Rule).* A recursive proto-rule is a proto-rule (as defined in Definition 4.4) that corresponds to a path  $P$  in a recursive constraint graph. This path must satisfy the condition that for every qualified table  $T_i$  in  $P$ , if  $i > 0$ , then for every  $j \in [0, i - 1]$ , there exists a qualified table  $T_j$  in  $P$ .

The condition on the indexes of qualified attributes ensures that indexes start at 0 and are given consecutively. We can now convert paths in that graph to proto-rules and to TGDs:

*Definition 5.4 (Recursive split proto-rule).* A recursive proto-rule is a proto-rule obtained from a recursive constraint graph by Definition 4.4. A *dangling variable* of a proto-rule is a variable that appears at most once. A recursive proto-rule is *non-redundant*, if it does not contain two atoms that (1) have the same table name, irrespective of table occurrence, and (2) agree on all non-dangling variables. Non-redundant proto-rules can be split and translated into TGDs according to Definition 4.7.

Redundant proto-rules take the form  $Lineage_1(\text{parent} = x) \Rightarrow Lineage_2(\text{parent} = x)$ . Such proto-rules can be refined in Algorithm 1, but they will not be used to generate TGDs. We thus obtain:

*Definition 5.5 (Matilda-N).* Matilda- $N$  (for a natural number  $N > 0$ ) is Algorithm 3, in which the constraint graph is replaced by a recursive constraint graph of degree  $N$ .

## 5.2 Theoretical Analysis

*Duplicate control.* MATILDA- $N$  can yield the same TGD twice.

EXAMPLE 12 (DUPLICATES.). *The following proto-rules are equal:*

### Proto-Rule 1

$$Lineage_0.\text{parent} = Lineage_1.\text{parent} \quad (1)$$

$$Lineage_0.\text{child} = Marriage_0.\text{partner1} \quad (2)$$

$$Lineage_1.\text{child} = Marriage_0.\text{partner2} \quad (3)$$

### Proto-Rule 2

$$Lineage_0.\text{parent} = Lineage_1.\text{parent} \quad (4)$$

$$Lineage_0.\text{child} = Marriage_0.\text{partner2} \quad (5)$$

$$Lineage_1.\text{child} = Marriage_0.\text{partner1} \quad (6)$$

*These proto-rules look different but give rise to two logically equivalent TGDs:*

$$\begin{aligned} & Marriage_0(\text{partner1}=b, \text{partner2}=c) \wedge Lineage_1(\text{parent}=a, \text{child}=c) \\ & \Rightarrow Lineage_0(\text{parent}=a, \text{child}=b) \end{aligned} \quad (7)$$

$$\begin{aligned} & Marriage_0(\text{partner1}=b, \text{partner2}=c) \wedge Lineage_0(\text{parent}=a, \text{child}=b) \\ & \Rightarrow Lineage_1(\text{parent}=a, \text{child}=c) \end{aligned} \quad (8)$$

This issue appears when the maximal TGD length is greater than two. To avoid yielding the same TGD twice, we cache the results and filter out duplicate TGDs. Some TGDs can imply each other (e.g.,  $\forall x : a(x) \Rightarrow c(x)$  implies  $\forall x : a(x) \wedge b(x) \Rightarrow c(x)$ ). One could thus believe that it would be sufficient to mine only one of them. However, the first TGD will likely have a higher coverage, while the second one will likely have a higher confidence. We have already discussed that the coverage and confidence of one TGD cannot be directly computed from the coverage and confidence of the other (below Example 4). Hence, we mine both TGDs and let the user choose.

*Completeness.* MATILDA-N is complete in the following sense:

**PROPOSITION 5.6 (COMPLETENESS OF MATILDA-N).** *Fix  $N \geq 1$  and a maximal number of atoms  $\theta \in \mathbb{N}$ . For any  $N$ -recursive TGD*

$$\tau = \forall \bar{x}, \bar{y} \phi(\bar{x}, \bar{y}) \implies \exists \bar{z} \psi(\bar{x}, \bar{z}), \quad |\phi \cup \psi| \leq \theta,$$

*MATILDA-N outputs a TGD that is logically equivalent to  $\tau$ .*

**PROOF. (i) Path enumeration.** Since  $\tau$  is  $N$ -recursive, every atom uses a qualified table  $T_i$  with  $0 \leq i < N$  (Definition 5.1). Traversing the equalities of  $\tau$  in lexicographic order yields a path  $p_\tau$  in the  $N$ -recursive constraint graph  $G_N$  (Definition 5.2) that fulfills the index-density condition of Definition 5.3. Algorithm 1 (DFS) enumerates  $p_\tau$  as long as  $|p_\tau| \leq \theta$ .

**(ii) Proto-rule creation.** Line 4 of Algorithm 1 converts  $p_\tau$  into a proto-rule  $r_\tau$  whose atoms are exactly those of  $\tau$ ; non-redundancy carries over from  $\tau$  to  $r_\tau$ .

**(iii) Splitting.** Algorithm 2 iterates over all admissible body-head partitions of  $r_\tau$ ; one split  $S_\tau$  coincides with  $(\phi, \psi)$  (or its symmetric counterpart).

**(iv) Emission.** When Algorithm 3 (MATILDA-N, Definition 5.5) processes  $\langle r_\tau, S_\tau \rangle$ , the support test in line 10 succeeds, so the resulting TGD is yielded. Thus at least one rule equivalent to  $\tau$  is produced, proving completeness.  $\square$

*Implementation.* MATILDA is implemented in Python, *SQLAlchemy* for interaction with the databases and *psutil* for monitoring purposes. The choice of Python distinguishes MATILDA from most of the other current rule mining approaches, which are implemented in Java or Prolog [10, 45, 92, 94]. However, data scientists typically work with Python nowadays.

## 6 Experiments

The goal of our experimental evaluation of MATILDA is threefold: first, to demonstrate that our algorithm scales to databases that are out of reach for our competitors; second, to show that it mines all rules that competing methods mine; and third, to evaluate the portion of TGDs that go beyond inclusion dependencies and Horn rules. We also examine the scalability and adaptability of MATILDA in various ablation studies.

### 6.1 Design Choices

Mining all TGDs faces a combinatorial explosion (Example 6). We therefore constrain both *structure* and *search*. Structurally, we restrict to connected patterns, disallow constants/arithmetic in rules, and require joins to follow a schema-level joinability relation. On the search side, we avoid enumerating a global partial order and instead prune candidates by joinability, minimum support and confidence, and a user-specified upper bound  $k$  on atoms per rule (in line with prior work [10, 45, 95]).

*Foreign-Key Joinability.* We allow joins only when the columns are connected by a declared foreign key in the database. This makes the join map small and easy to build from the schema. It also matches how systems like AMIE treat relations with clear “points to” links. Because foreign key networks are usually sparse, this choice quickly reduces the number of joins we need to try. Most importantly, foreign keys capture the designer’s intent about which rows refer to the same real-world thing (for example, `Order.customer_id` points to `Customer.id`). As a result, our joins are meaningful, and we avoid guesswork based on overlapping values or similar strings.

*Closed-World Semantics and NULLs.* We adopt the Closed World Assumption: Any ground atom not stored in the database is treated as *false*. Tuples containing NULL are excluded from matching and aggregation; consequently, we neither instantiate TGDs with incomplete bindings nor count partial instances. This ensures that measured support and confidence rest on fully grounded facts.

## 6.2 Setup

*Databases.* MATILDA is designed to take a database as input. We chose the relational databases from the relational data project<sup>4</sup> [46], which are well-documented and widely used in academic research [17, 41, 76]. We removed databases that could not be converted from the MariaDB format to SQLite using the tool `mysql-to-sqlite3`<sup>5</sup>, and databases that contained image data. The remaining databases are shown in Table 1, covering a wide variety of table arities, sizes, and number of tables. We consider two attributes joinable if one is a foreign key of the other.

Database	Tables	Rows	Col's	Triples	FK	Database	Tables	Rows	Col's	Triples	FK
Biodegradability	5	21875	14	48998	0.123	Mesh	29	2700	37	1704	0.345
Bupa	9	2762	16	2069	0.085	MuskSmall	2	554	170	79584	0.567
Carcinogenesis	6	27570	23	92486	0.456	mutagenesis	3	10324	14	36241	0.876
ccs	6	422868	29	473942	0.789	nations	3	11004	118	33171	0.234
CDESchools	3	29481	90	872104	0.234	NBA	5	1221	72	17446	0.456
Chess	2	2052	45	28766	0.678	NCAA	9	201555	106	2390826	0.678
CiteSeer	3	113760	6	3311	0.345	Pima	9	6912	18	13056	0.789
classicmodels*	8	3864	59	14071	0.567	PTE	38	29762	76	89120	0.345
ConsumerExp.	3	2241548	24	15479875	0.876	pubs*	11	255	64	945	0.567
CORA	3	57884	6	2707	0.234	Pyrimidine	2	296	13	2071	0.876
Countries	4	21118	67	516839	0.456	SAP*	5	4130447	45	15366486	0.234
CraftBeer	2	2968	11	16133	0.678	SAT	36	19867	69	560	0.456
cs*	8	563	43	3279	0.789	SFScores	3	66153	25	370675	0.678
Dallas	3	812	27	6623	0.345	Shakespeare	4	35234	19	149609	0.789
DCG*	2	8258	5	8257	0.567	Toxicology	4	49239	11	124040	0.345
Dunur	17	440	34	275	0.876	tpcc*	9	593933	93	3893757	0.567
Elti	11	1352	22	1080	0.234	UTube	2	2735	7	7110	0.876
Hockey	22	96403	300	264268	0.456	Walmart	4	4628497	27	4986950	0.234
KRK*	1	1000	8	7000	0.678	WebKP	3	80592	6	876	0.456
medical	3	15899	64	263023	0.789	world	3	5411	24	21629	0.678

Table 1. Database statistics. *Triples* is the number of *distinct* RDF triples obtained by applying the W3C *Direct Mapping* from relational data to RDF (RDB2RDF DM), counting only non-NULL attribute values; we include class assertions (`rdf:type`) and attribute/object-property triples and deduplicate identical triples before counting. *FK* is, for each database, the average over tables of the ratio of declared foreign-key columns to total columns, i.e.,  $\frac{1}{|T|} \sum_{t \in T} \frac{\#FK(t)}{\#cols(t)}$ ; higher values indicate more referentially dense schemas. An asterisk (\*) marks synthetic data. The *Triples* column reflects the size of the AMIE knowledge base constructed from the direct mapping.

*Experimental setup.* All experiments were conducted on a machine equipped with an 11th-generation Intel Core i7-1165G7 processor (x86\_64 architecture), running at a 2.80 GHz base frequency with a maximum turbo frequency of 4.70 GHz. The CPU features 4 physical cores and 8 logical threads (Hyper-Threading enabled), with a cache hierarchy comprising 192 KiB of L1d cache, 128 KiB of L1i cache, 5 MiB of L2 cache, and 12 MiB of L3 cache. The system is equipped with 32 GB of DDR4 RAM. The software environment is Ubuntu 24.04.3 LTS (Linux kernel 6.8.0-90-generic).

*Choice of DBMS.* We use `SQLITE` because it does not require the installation and configuration of a server to replicate our experiments.

*Competitors.* We compare our approach to the following competitors: Spider (Metanome project)[94] is a state-of-the-art inclusion dependency mining system; AMIE 3[45] is the newest incarnation of the AMIE Horn rule mining system; and Popper [10] is, at the time of writing, the fastest ILP system – ILP systems being particularly notable for mining Horn rules. All systems (Popper, Spider, AMIE, and MATILDA) mine both patterns with a 100% confidence and approximate patterns with a confidence of less than 100%. Spider does not provide confidence or support metrics, Popper reports only support, and AMIE and MATILDA output both.

<sup>4</sup><https://relational-data.org/>

<sup>5</sup><https://github.com/techohouse/mysql-to-sqlite3>

Database	Popper			Spider			AMIE 3			MATILDA		
	j'ble	results	time	j'ble	results	time	j'ble	results	time	results	Cov.	time
<b>Biodegradability</b>	0	2	57m:52s ± 10m	4	7	0.99s ± 0.02s	-	2	2.65s ± 0.13s	4	100%	0.05s ± 1 $\mu$ s
<b>Bupa</b>	1	24	1.05s ± 0.01s	6	58	0.84s ± 0.01s	-	0	0.94s ± 0.05s	9	100%	0.08s ± 1 $\mu$ s
<b>Carcinogenesis</b>	0	2	2.02s ± 0.04s	1	25	1.12s ± 0.03s	0	8	2.92s ± 0.08s	2	100%	0.04s ± 1 $\mu$ s
<b>ccs</b>	-	-	timeout	-	233	2.34s ± 0.04s	0	4	15.70s ± 0.59s	0	-	0.02s ± 1 $\mu$ s
<b>CDESchools</b>	-	-	timeout	2	55	6.85s ± 0.66s	-	-	timeout	4	100%	0.09s ± 1 $\mu$ s
<b>Chess</b>	-	-	timeout	1	46	1.09s ± 0.03s	0	-	timeout	2	100%	0.03s ± 1 $\mu$ s
<b>CiteSeer</b>	-	0	2m:34s ± 1m:32s	2	2	1.15s ± 0.03s	-	0	3.20s ± 0.11s	2	100%	0.18s ± 1 $\mu$ s
<b>classicmodels</b>	-	-	timeout	6	132	1.04s ± 0.02s	0	31	1.91s ± 0.06s	14	100%	0.15s ± 1 $\mu$ s
<b>ConsumerExp</b>	-	-	timeout	0	31	16.34 ± 0.97s	-	-	timeout	0	-	1 $\mu$ s ± 1 $\mu$ s
<b>CORA</b>	0	2	9.83s ± 2.56s	4	6	1.03s ± 0.01s	-	0	2.15s ± 0.14s	6	100%	0.12s ± 1 $\mu$ s
<b>Countries</b>	-	-	timeout	-	0	0.52s ± 0.01s	0	0	5m:1.60s	-	mem-out	-
<b>CraftBeer</b>	0	2	17.07s ± 1.66s	1	2	0.93s ± 0.04s	0	4	1.62s ± 0.06s	2	100%	0.03s ± 1 $\mu$ s
<b>cs</b>	0	-	timeout	8	32	0.83s ± 0.02s	0	35	1.01s ± 0.05s	27	100%	0.19s ± 1 $\mu$ s
<b>Dallas</b>	-	-	timeout	-	8	1.17s ± 0.02s	0	4	1.08s ± 0.03s	0	-	0.01s ± 1 $\mu$ s
<b>DCG</b>	-	0	0.55s ± 0.00s	-	4	0.94s ± 0.02s	-	0	0.94s ± 0.02s	0	-	0.01s ± 1 $\mu$ s
<b>Dunur</b>	1	35	3.66s ± 0.41s	20	164	0.96s ± 0.02s	-	0	0.59s ± 0.01s	40	100%	0.37s ± 1 $\mu$ s
<b>Elti</b>	2	37	1.81s ± 0.01s	12	78	0.92s ± 0.02s	-	0	0.68s ± 0.01s	24	100%	0.23s ± 1 $\mu$ s
<b>Hockey</b>	-	-	mem-out	-	11172	7.86s ± 0.3124	-	-	timeout	-	timeout	-
<b>KRK</b>	-	0	11.11s ± 1.41s	-	30	0.85s ± 0.02s	0	58	1.49s ± 0.04s	0	-	0.01s ± 1 $\mu$ s
<b>medical</b>	-	-	timeout	1	116	2.47s ± 0.04s	0	-	timeout	4	100%	0.08s ± 1 $\mu$ s
<b>Mesh</b>	-	6	8.00s ± 1.27s	0	64	0.87s ± 0.02s	-	0	0.60s ± 0.01s	8	100%	0.06s ± 1 $\mu$ s
<b>MuskSmall</b>	-	-	timeout	2	62	1.88s ± 0.01s	0	-	timeout	2	100%	0.06s ± 1 $\mu$ s
<b>mutagenesis</b>	0	1	4m:34.17s	2	6	0.97s ± 0.03s	0	5	1.88s ± 0.13s	2	100%	0.04s ± 1 $\mu$ s
<b>NBA</b>	-	-	timeout	8	392	1.35s ± 0.07s	0	-	timeout	25	100%	0.20s ± 1 $\mu$ s
<b>nations</b>	-	-	timeout	2	12444	3.12s ± 0.06s	0	-	timeout	2	100%	0.08s ± 1 $\mu$ s
<b>NCAA</b>	-	-	timeout	10	879	4.63s ± 0.08s	0	-	timeout	28	100%	2.85s ± 0.03s
<b>PTE</b>	6	12	35m:22s ± 3m:1s	22	300	1.58s ± 0.04s	-	0	3.11s ± 0.10s	33	100%	0.29s ± 0.01s
<b>Pima</b>	0	12	2.00s ± 0.01s	10	72	0.92s ± 0.01s	-	0	1.40s ± 0.04s	10	100%	0.10s ± 1 $\mu$ s
<b>Pyrimidine</b>	-	-	timeout	-	26	0.94s ± 0.04s	0	-	timeout	0	-	0.01s ± 1 $\mu$ s
<b>SAP</b>	-	-	timeout	1	41	18.49s ± 0.71s	-	-	timeout	2	100%	0.45s ± 0.01s
<b>SAT</b>	4	10	2m:46s ± 2.29s	-	1639	1.51s ± 0.0435	-	0	2.70s ± 0.17s	20	100%	0.21s ± 1 $\mu$ s
<b>SFScores</b>	-	-	timeout	-	3	1.67s ± 0.08s	0	23	6.10s ± 0.24s	0	-	0.01s ± 1 $\mu$ s
<b>Shakespeare</b>	-	0	6.48s ± 0.01s	2	13	1.67s ± 0.04s	0	10	3.99s ± 0.20s	2	100%	0.03s ± 1 $\mu$ s
<b>Toxicology</b>	0	6	6.44s ± 0.63s	2	10	1.16s ± 0.02s	0	5	4.62s ± 0.16s	4	100%	0.07s ± 1 $\mu$ s
<b>UTube</b>	-	0	0.49s ± 0.00s	0	3	0.87s ± 0.02s	-	0	1.00s ± 0.04s	2	100%	0.04s ± 1 $\mu$ s
<b>Walmart</b>	1	4	3m:29s ± 48s	2	88	10.05s ± 0.58s	-	95	3m:46.16s ± 7.66s	2	100%	0.05s ± 1 $\mu$ s
<b>WebKP</b>	0	2	7.78s ± 1.70s	4	6	1.23s ± 0.01s	0	0	2.95s ± 0.26s	6	100%	0.18s ± 1 $\mu$ s
<b>tpcc</b>	-	-	timeout	12	639	11.51s ± 0.10s	0	17	2m:0.45s ± 4.39s	24*	100%	03h:15m ± 0.44m
<b>world</b>	-	-	timeout	0	4	0.95s ± 0.02s	0	15	1.65s ± 0.09s	2	100%	0.04s ± 1 $\mu$ s

Table 2. Comparison of MATILDA against other methods in terms of the number of results, execution time, and coverage. Reported runtimes are the mean over 5 independent runs with the dispersion shown as  $\mu \pm \Delta$ . A value of “-” indicates that the corresponding data cannot be computed. The “j’ble” column gives the number of TGDs that join only joinable columns. The “Cov.” column gives the percentage of the joinable TGDs of the competitors that MATILDA mines. In the **MATILDA results** column, the symbol \* (on *tpcc*) indicates a large-scale run (about 1M rows) executed with a constraint of at most two tables per candidate TGD; see Appendix B (*Bupa-1M*) for additional scalability results.

Each of our competitors required slight adaptations of our dataset. For Spider, we converted the database into CSV files, one for each table. We configured the Metanome framework to be used with the command-line interface extension. For AMIE, all tables had to be converted to binary relations. We used a direct mapping<sup>6</sup> to automatically convert the database into a knowledge base of such binary relations.

For Popper, each table in the database is translated into three Prolog files:

- *bias.pl*: Encodes the database schema (including table arities) and configuration settings.
- *bk.pl*: Represents the entire database as background knowledge.
- *exs.pl*: Contains examples drawn from the table that has been selected as the target (i.e., the head of the generated Horn rules).

We thank the authors of Popper for their assistance with this transformation process.

To cover all potential rules, we set the maximum number of atoms in a rule body (configured in *bias.pl*) to the total number of tables in the database. For Popper to leverage joinability constraints,

<sup>6</sup><https://www.w3.org/TR/rdb-direct-mapping/>

Database	Rule	Interpretation
NCAA	$\forall x_0 : \text{Team}_0(\text{TeamId}=x_0) \Rightarrow \exists z_0 : \text{Actions}_0(\text{GameId}=z_0) \wedge \text{Game}_0(\text{GameId}=z_0, \text{Team2Id}=x_0)$	Every team $x_0$ must appear as the second team ( $\text{Team2Id} = x_0$ ) in at least one game $z_0$ , together with an action for that game.
Dunur	$\forall x_0 : \text{person}_0(\text{name}=x_0) \Rightarrow \exists z_0 : \text{brother}_0(\text{name1}=x_0, \text{name2}=z_0) \wedge \text{person}_1(\text{name}=z_0)$	Every person $x_0$ has at least one brother $z_0$ who is also a person (non-perfect rule).
University	$\forall x_0 : \text{student}_0(\text{student\_id}=x_0) \Rightarrow \exists z_0 : \text{RA}_0(\text{prof\_id}=z_0, \text{student\_id}=x_0) \wedge \text{prof}_0(\text{prof\_id}=z_0)$	Every student $x_0$ has at least one professor $z_0$ such that $x_0$ is a research assistant to $z_0$ , and $z_0$ is a valid professor.
CORA	$\forall x_0 : \text{paper}_0(\text{paper\_id}=x_0) \Rightarrow \exists z_0 : \text{cites}_0(\text{cited\_paper\_id}=x_0, \text{citing\_paper\_id}=z_0) \wedge \text{paper}_1(\text{paper\_id}=z_0)$	Every paper $x_0$ is cited by at least one paper $z_0$ .
PTE	$\forall x_0 : \text{pte\_drug}_0(\text{drug\_id}=x_0) \Rightarrow \exists z_0 : \text{pte\_active}_0(\text{drug\_id}=x_0, \text{is\_active}=z_0) \wedge \text{pte\_number}_0(\text{binary}=z_0)$	Each drug $x_0$ has an active flag $z_0$ (in $\text{pte\_active}_0$ ), and that same flag appears as a binary value in $\text{pte\_number}_0$ .
UW	$\forall x_0 : \text{course}(\text{course\_id}=x_0) \Rightarrow \exists z_0 : \text{advisedBy}(\text{p\_id\_dummy}=z_0) \wedge \text{taughtBy}(\text{p\_id}=z_0, \text{course\_id}=x_0)$	Every course $x_0$ has at least one person $z_0$ who both advises and teaches it.
Same-gen	$\forall x_0 : \text{same\_gen}(\text{name2}=x_0) \Rightarrow \exists z_0 : \text{parent}(\text{name1}=x_0, \text{name2}=z_0) \wedge \text{person}(\text{name}=z_0)$	Entities marked as same generation have a parent-child relationship, revealing generational or hierarchical structures.

Table 3. Examples of TGDs that only MATILDA mines. Only the first rule is a “perfect” one, whose constituent Horn rule has a confidence of 100%.

we remove any table from both *bias.pl* and *exs.pl* unless it shares at least one node in the constraint graph with the selected head table. In this context, “sharing a node” means that there is at least one attribute linking the two tables in the constraint graph. For instance, in our example, if no node in the constraint graph includes both an attribute from the “Marriage” table and an attribute from the “Lineage” table (the selected head), then all data from the “Marriage” table would be removed. The detailed parameter settings are in Appendix A. We note that all competitors have access to the same information (tables, foreign keys, etc.), yet not all of them are designed to make use of it.

*Server Settings.* All experiments were conducted on a single machine (Intel Core i7-1165G7, 32 GB RAM) running Ubuntu 24.04.3 LTS (kernel 6.8.0-90-generic). All methods were executed in Docker containers to ensure a consistent and reproducible environment. Each run used the full compute of the machine (all CPU threads), with memory capped at 30 GB to prevent system freezes.

### 6.3 Results

*Effectiveness.* Table 2 shows the results of all mining systems on all databases. The Popper system fails on half of the databases with a memory overflow. This is not surprising, as ILP systems are typically designed for smaller datasets.

Spider runs on nearly all databases. It finds many inclusion dependencies, and in most databases the ones it reports are correct (i.e., one column is indeed a subset of the other column). It finds only inclusion dependencies, not Horn rules (let alone TGDs).

AMIE runs on most databases, but finds only very few rules. This has two reasons: First, decomposing relations into binary relations increases rule length. For example,  $\text{Married}(a,b) \wedge \text{Residence}(a,l,s) \Rightarrow \text{Residence}(b,l,s)$  has 2 body atoms on the relational schema but inflates to 5–7 binary atoms after

triple encoding. Many such rules then exceed AMIE’s rule-length budget  $L$ ; increasing  $L$  causes a combinatorial explosion in candidates and joins, so these rules remain effectively unreachable in practice. Second, the conversion to the triple format (id, relation, value) hinders AMIE from mining meaningful rules, as it restricts the system to patterns that directly associate an ID with an attribute.

MATILDA runs on all databases except *Hockey*. This is because the induced constraint graph on this dataset is extremely large (due to a dense foreign-key graph and many qualified pairs). On all other datasets, MATILDA reproduces all joinable patterns mined by the other approaches. Since it mines patterns beyond inclusion dependencies and simplistic AMIE rules, MATILDA produces about 50% more patterns overall. Despite this, MATILDA runs roughly in the same time as the other approaches. We thus conclude that MATILDA is the only system capable of finding patterns beyond inclusion dependencies in larger databases.

*Coverage.* MATILDA mines all joinable correct inclusion dependencies that Spider mines, and all joinable rules that Popper mines. MATILDA does not mine all the rules that AMIE mines. This is because AMIE has no mechanism to account for joinability (even if we wanted to provide this information to AMIE). Therefore, most rules it mines are spurious correlations of the form  $relation(id=x_0, age=x_1) \Rightarrow relation(id=x_0, count=x_1)$  (objects of type *relation* have the same *age* as their *count*). Furthermore, manual inspection shows that, due to the decomposition into binary relations, almost all of AMIE’s rules (99.91%) revolve around joins across a single table. However, some of them can be considered useful (such as the rule that the number of officers sent to an incident often equals the number of involved people). We show in our ablation study<sup>7</sup> that if we consider all columns that share a value joinable, MATILDA can mine all of AMIE’s rules.

*Complex TGDs.* Out of the 1050 rules that MATILDA mined on all databases, 8% are TGDs that have more than one atom in the head (linked by an existential variable), i.e., patterns that our competitors cannot mine by design. Some examples are shown in Table 3. Of these, 44% are “perfect”, i.e., they are of the form  $\forall \vec{x}, \vec{y} : \phi(\vec{x}, \vec{y}) \Rightarrow \exists \vec{z} : a(\vec{y}, \vec{z}) \wedge b(\vec{y}, \vec{z})$ , where  $a(\vec{y}, \vec{z})$  and  $b(\vec{y}, \vec{z})$  are atoms and there exists also a rule  $\forall \vec{y}, \vec{z} : a(\vec{y}, \vec{z}) \Rightarrow b(\vec{y}, \vec{z})$  with a confidence of 1. Such rules are correct and interesting, because they confirm type constraints, e.g., the first example in Table 3. However, they could in principle be reconstructed from the individual Horn rules of which they are composed, in the way outlined in Section 2.1. In future work, these rules could be anticipated and excluded from the mining. The remaining 56% of non-Horn rules express type constraints that do not hold with a confidence of 100%, e.g., the other examples in Table 3, meaning that their confidence cannot be computed from the confidences of the constituent rules (Section 3.2). These non-perfect rules are all of the same shape: They vary mainly in the relation names, e.g., sister instead of brother. The observation that many non-perfect rules share the same structure and differ mainly in relation names suggests that the mined TGDs capture underlying structural regularities of the schema rather than isolated correlations. In particular, these patterns reveal families of relations that play analogous roles in joins, indicating latent equivalence classes at the schema level. This may reflect either intentional modeling choices (e.g., semantically distinct but structurally similar relations) or redundancies that may require further investigation by the data modeler. Indeed, such regularities are useful for schema understanding and validation, as they can highlight inconsistencies when one relation deviates from the common pattern, and may guide schema refinement by suggesting abstractions or unifications across relations.

<sup>7</sup><https://github.com/Fran-cois/MATILDA>

## 6.4 Ablation studies

We conduct several ablation studies. We chose the BUPA dataset because it is the only one on which MATILDA runs across all ablation settings. For space reasons, the ablation studies are in our GitHub repository<sup>7</sup>, and we only summarize the results here.

*Joinability.* We ablate three joinability regimes: *FK-only* (default), *rule-based*, and *full*. *FK-only* restricts joins to schema-declared foreign keys. *Rule-based* permits joins only for attributes of the same declared type (numerics excluded) with Jaccard similarity  $\geq 50\%$  or mutual coverage  $\geq 70\%$ . *Full* joinability treats every attribute pair as joinable (e.g.,  $Lineage(parent = x_0) \Rightarrow Residence(state = x_0)$ ), effectively disabling pruning. On BUPA, varying recursion depth  $d \in \{1, 2, 3\}$  keeps *FK-only* runtime essentially flat (variation  $\approx 0.01$  s), while *rule-based* and *full* increase sharply in time and result count – supporting our choice of limiting joinability to foreign keys.

*Sampling rows.* We vary the row-sampling rate on BUPA and measure runtime and result count at recursion depths  $d \in \{1, 2, 3\}$ . We observe a roughly linear drop in discovered rules as rows are removed, but recursion depth dominates: on the log–log plot, the curves are primarily ordered by  $d$  rather than by the sampling rate. Lower sampling (e.g., 10%) runs faster but finds fewer rules; higher sampling (e.g., 90%) increases computation and coverage. Overall, row sampling affects throughput, while recursion depth is the main driver of result growth.

*Sampling columns.* Reducing attributes has a much larger effect than reducing rows. Rule counts can drop by about  $10^3$  under column pruning (versus about  $10^2$  for row pruning), and the slowdown intensifies as  $d$  increases. Random removal introduces small irregularities (e.g., 10% vs. 40%) because some attributes are pivotal for rule formation. In practice, attribute count is the key complexity lever, and column pruning trades recall for speed more sharply than row sampling.

*Coverage of AMIE Rules.* To reproduce AMIE-style rules, we run MATILDA with `joinable=ALL`, `recursion=2`, `max_table=2`, `max_var=1`. In this setting, MATILDA is up to 40× slower than AMIE because the constraint graph explodes without joinability constraints, but it returns up to 100× more patterns.

## 7 Conclusion

We presented MATILDA, an approach for mining dependencies on relational databases. The key idea in dealing with the exponential size of the search space is to represent possible joins in a constraint graph. Traversing this graph is still computationally expensive, but it gives more opportunities for pruning. Through extensive experiments on multiple benchmark datasets, we have demonstrated not only that MATILDA scales to databases that are out of reach for our competitors, but also that it mines all joinable patterns that competing approaches find, plus a few more complex TGDs.

Future work could extend MATILDA to mine equality-generating dependencies (EGDs). These would allow mining keys, functional dependencies, and conditional functional dependencies.

## Acknowledgements

This work has been partially supported the 3IA Côte d’Azur Investments in the IA-cluster project managed by the National Research Agency (ANR) with the reference number ANR-23-IACL-0001. The work was also supported by Hi! PARIS and ANR/France 2030 program (ANR-23-IACL-0005).

## References

- [1] E. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* (1970).
- [2] W. Cohen. 1994. Grammatically biased learning: learning logic programs using an explicit antecedent description language. *Artificial Intelligence* (1994).

- [3] W. Cohen. 1995. Inductive specification recovery: Understanding software by learning from example behaviors. *International Conference on Automated Software Engineering* (1995).
- [4] F. de Marchi et al. 2002. Efficient Algorithms for Mining Inclusion Dependencies. In *International Conference on Extending Database Technology (EDBT)*.
- [5] A. Deutsch. 2009. *FOL Modeling of Integrity Constraints (Dependencies)*.
- [6] A. Alexandrov et al. 2014. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal* (2014).
- [7] A. Agostini et al. 2023. A Novel Integration of Data-Driven Rule Generation and Computational Argumentation for Enhanced Explainable AI. *Journal of AI Research* (2023).
- [8] A. Bordes et al. 2013. Translating Embeddings for Modeling Multi-relational Data. *NIPS* (2013).
- [9] A. Cal et al. 2008. Taming the Infinite Chase: Query Answering under Expressive Relational, Constraints. In *Knowledge Representation and Reasoning*.
- [10] A. Cropper et al. 2020. Learning programs by learning from failures.
- [11] A. Cropper et al. 2020. Turning 30: New Ideas in Inductive Logic Programming.
- [12] A. Heise et al. 2013. Scalable Discovery of Unique Column Combinations. *Proc. VLDB Endow.* (2013).
- [13] A. Ouaddah et al. 2020. A Schema Integration Approach for Big Data Analysis. *International Journal of Intelligent Systems and Applications* (2020).
- [14] A. Rostin et al. 2009. A Machine Learning Approach to Foreign Key Discovery.. In *International Workshop on the Web and Databases (WebDB)*.
- [15] A. Sharma et al. 2020. DoWhy: An End-to-End Library for Causal Inference. *arXiv:2011.04216* (2020).
- [16] A. Vasoya et al. 2016. Mining of Association Rules on Large Database Using Distributed and Parallel Computing. *Procedia Computer Science* (2016).
- [17] B. Levin et al. 1999. PKDD'99 discovery challenge—financial domain. In *Workshop Notes on Discovery Challenge*.
- [18] B. Steenwinckel et al. 2022. INK: knowledge graph embeddings for node classification. *Data Min. Knowl. Discov.* (2022).
- [19] C. Beeri et al. 1977. A complete axiomatization for functional and multivalued dependencies in database relations. In *ACM SIGMOD international conference on Management of data*.
- [20] C. Leckie et al. 1998. Inductive learning of search control rules for planning. *Artificial Intelligence* (1998).
- [21] C. Meilicke et al. 2019. Anytime Bottom-Up Rule Learning for Knowledge Graph Completion. In *International Joint Conference on Artificial Intelligence*.
- [22] C. Meilicke et al. 2019. An introduction to AnyBURL. In *Advances in Artificial Intelligence*.
- [23] D. Arpit et al. 2023. Salesforce CausalAI Library: A Fast and Scalable Framework for Causal Analysis of Time Series and Tabular Data. *CoRR* (2023).
- [24] D. Symeonidou et al. 2014. SAKey: Scalable Almost Key Discovery in RDF Data. In *International Semantic Web Conference*.
- [25] D. Symeonidou et al. 2017. VICKEY: Mining Conditional Keys on Knowledge Bases. In *International Workshop on the Semantic Web*.
- [26] E. FARJANA et al. 2022. Competent Triple Identification for Knowledge Graph Completion under the Open-World Assumption. *IEICE Transactions on Information and Systems* (2022).
- [27] E. Pena et al. 2019. Discovery of approximate (and exact) denial constraints. *Vldb Endowment* (2019).
- [28] E. Pena et al. 2021. Fast detection of denial constraint violations. *VLDB Endowment* (2021).
- [29] E. Pena et al. 2022. Fast Algorithms for Denial Constraint Discovery. *VLDB Endowment* (2022).
- [30] E. Zhu et al. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables, in Data Lakes. In *International Conference on Management of Data*.
- [31] F. Dursch et al. 2019. Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms. *ACM International Conference on Information and Knowledge Management* (2019).
- [32] F. Suchanek et al. 2019. Knowledge Representation and Rule Mining in Entity-Centric Knowledge Bases. In *RW invited paper*.
- [33] F. Yang et al. 2017. Differentiable Learning of Logical Rules for Knowledge Base Completion. *CoRR* (2017).
- [34] G. Mecca et al. 2009. Core Schema Mappings. In *ACM SIGMOD International Conference on Management of Data*.
- [35] H. Mannila et al. 1994. Efficient algorithms for discovering association rules. In *AAAI Workshop on Knowledge Discovery in Databases (KDD)*.
- [36] H. Zhang et al. 2015. Existential Rule Languages with Finite Chase: Complexity and Expressiveness. *AAAI Conference on Artificial Intelligence* (2015).
- [37] H. Zhang et al. 2021. Characterizing the Program Expressive Power of Existential Rule Languages.
- [38] J. Bauckmann et al. 2007. Efficiently Detecting Inclusion Dependencies. In *International Conference on Data Engineering*.
- [39] J. Baget et al. 2011. On rules with existential variables: Walking the decidability line. *Artificial Intelligence* (2011).

- [40] J. Baget et al. 2011. Walking the complexity lines for generalized guarded existential rules. In *International Joint Conference on Artificial Intelligence*.
- [41] J. Cheng et al. 2002. KDD Cup 2001 report. *ACM SIGKDD Explorations Newsletter* (2002).
- [42] J. Jøsang et al. 2022. On the Effectiveness of Knowledge Graph Embeddings: a Rule Mining Approach.
- [43] J. Kossmann et al. 2022. Data dependencies for query optimization: a survey. *The VLDB Journal* (2022).
- [44] J. Liu et al. 2012. *IEEE Trans. Knowl. Data Eng.* (2012).
- [45] J. Lajus et al. 2020. Fast and Exact Rule Mining with AMIE 3. *The Semantic Web* (2020).
- [46] J. Motl et al. 2015. The CTU Prague Relational Learning Repository.
- [47] J. Motl et al. 2017. Foreign Key Constraint Identification in Relational Databases.. In *ITAT*.
- [48] J. Quinlan et al. 1995. Induction of logic programs: FOIL and related systems. *New Generation Computing* (1995).
- [49] J. Szlichta et al. 2016. Effective and Complete Discovery of Order Dependencies via Set-based Axiomatization.
- [50] J. Volker et al. 2011. Statistical schema induction. In *Extended Semantic Web Conference*.
- [51] J. Wogulis et al. 1993. A Methodology for Evaluating Theory Revision Systems: Results with Audrey II. *International Joint Conference on Artificial Intelligence* (1993).
- [52] J. Zelle et al. 1993. Combining FOIL and EBG To Speed-up Logic Programs. *International Joint Conference on Artificial Intelligence* (1993).
- [53] J. Zelle et al. 1994. Combining Top-down and Bottom-up Techniques in Inductive Logic Programming. *International Conference on Machine Learning* (1994).
- [54] K. Ali et al. 1993. HYDRA: A Noise-tolerant Relational Concept Learning Algorithm. In *International Joint Conference on Artificial Intelligence*.
- [55] L. Bellomarini et al. 2022. Exploiting the Power of Equality-Generating Dependencies in Ontological Reasoning. *Proc. VLDB Endow.* (2022).
- [56] L. Caruccio et al. 2016. Relaxed Functional Dependencies—A Survey of Approaches. *IEEE Trans. Knowl. Data Eng.* (2016).
- [57] L. Caruccio et al. 2021. Discovering Relaxed Functional Dependencies Based on Multi-Attribute, Dominance. *IEEE Trans. Knowl. Data Eng.* (2021).
- [58] L. Galarraga et al. 2013. Mining Rules to Align Knowledge Bases. In *Workshop on Automated Knowledge Base Construction*.
- [59] L. Galarraga et al. 2015. Fast Rule Mining in Ontological Knowledge Bases with AMIE+. *VLDB J.* (2015).
- [60] L. Greeshma et al. 2016. Unique Constraint Frequent Item Set Mining. In *IEEE International Conference on Advanced Computing (IACC)*.
- [61] L. Popa et al. 2002. Translating Web Data. In *International Conference on Very Large Data Bases*.
- [62] M. Atencia et al. 2012. Keys and Pseudo-Keys Detection for Web Datasets Cleansing and Interlinking. In *EKAU: Knowledge Engineering and Knowledge Management*.
- [63] M. Bowling et al. 2006. Automated Action Selection Using Reinforcement Learning. *Autonomous Agents and Multi-Agent Systems* (2006).
- [64] M. Doan et al. 2024. Data Quality Awareness: A Journey from Traditional Data Management to Data Science Systems. *arXiv preprint* (2024).
- [65] M. Gyssens et al. 1984. On the decomposition of join dependencies. In *3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*.
- [66] M. Nickel et al. 2011. A Three-Way Model for Collective Learning on Multi-Relational Data. *International Conference on Machine Learning* (2011).
- [67] M. Saeed et al. 2021. RuleBERT: Teaching Soft Rules to Pre-Trained Language Models. In *Conference on Empirical Methods in Natural Language Processing*.
- [68] M. Shen et al. 2022. Fast Accurate Discovery of Tuple Inclusion Dependencies. In *IEEE International Conference on Smart Computing (SMARTCOMP)*.
- [69] N. Kouagou et al. 2024. Improving rule mining via embedding-based link prediction.
- [70] N. Pernelle et al. 2013. An automatic key discovery approach for data linking. *Journal of Web Semantics* (2013).
- [71] N. Radulović et al. 2021. Confident Interpretations of Black Box Classifiers (STACI). In *IJCNN*.
- [72] N. Shaabani et al. 2019. Incrementally updating unary inclusion dependencies in dynamic data. *Distributed and Parallel Databases* (2019).
- [73] P. Bernstein et al. 2011. Generic Schema Matching, Ten Years Later. *Proc. VLDB Endow.* (2011).
- [74] P. Bourhis et al. 2016. Guarded-Based Disjunctive Tuple-Generating Dependencies. *ACM Trans. Database Syst.*, Article 27 (2016).
- [75] P. Blobaum et al. 2022. DoWhy-GCM: An extension of DoWhy for causal inference in graphical causal models. *arXiv:2206.06821* (2022).
- [76] P. Sen et al. 2008. Collective classification in network data. *AI magazine* (2008).

- [77] Q. Wang et al. 2015. Propagating Dependencies under Schema Mappings: A Graph-Based Approach. In *International Database Engineering & Applications Symposium*.
- [78] Q. Zhao et al. 2003. Association Rule Mining: A Survey. In *Technical Report, Nanyang Technological University*.
- [79] Q. Zeng et al. 2014. QuickFOIL: scalable inductive logic programming. *VLDB Endowment* (2014).
- [80] R. Agrawal et al. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *International Conference on Very Large Data Bases*.
- [81] R. Agrawal et al. 1996. Fast discovery of association rules. *Advances in knowledge discovery and data mining* (1996).
- [82] R. Cappuzzo et al. 2025. Retrieve, Merge, Predict: Augmenting Tables with Data Lakes. *Trans. Mach. Learn. Res.* (2025).
- [83] R. Fagin et al. 2003. Data Exchange: Semantics and Query Answering. In *ICDT*.
- [84] R. Fernandez et al. 2018. Aurum: A Data Discovery System. In *International Conference on Data Engineering*.
- [85] R. Karegar et al. 2021. Efficient Discovery of Approximate Order Dependencies. *CoRR* (2021).
- [86] R. Pichler et al. 2011. The complexity of evaluating tuple generating dependencies. In *International Conference on Database Theory*.
- [87] R. Xiao et al. 2022. Fast Approximate Denial Constraint Discovery. *VLDB Endowment* (2022).
- [88] S. Abiteboul et al. 1995. *Foundations of databases*.
- [89] S. Castelo et al. 2021. Auctus: A Dataset Search Engine for Data Discovery and Augmentation. *Proc. VLDB Endow.* (2021).
- [90] S. Džeroski et al. 1993. Inductive learning in deductive databases. *IEEE Transactions on Knowledge and Data Engineering* (1993).
- [91] S. Kazemi et al. 2018. Simple embedding for link prediction in knowledge graphs. *Neural Information Processing Systems* (2018).
- [92] S. Ortona et al. 2018. Robust Discovery of Positive and Negative Rules in Knowledge Bases. In *ICDE*.
- [93] S. Ortona et al. 2018. RuDiK: Rule discovery in knowledge bases. In *VLDB*.
- [94] T. Papenbrock et al. 2015. Data Profiling with Metanome. *Proc. VLDB Endow.* (2015).
- [95] T. Papenbrock et al. 2015. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *Proc. VLDB Endow.* (2015).
- [96] T. Papenbrock et al. 2017. A hybrid approach for efficient unique column combination discovery. *Datenbanksysteme für Business, Technologie und Web* (2017).
- [97] T. Soru et al. 2015. ROCKER - A Refinement Operator for Key Discovery. In *WWW*.
- [98] T. Trouillon et al. 2016. Complex Embeddings for Simple Link Prediction. *International Conference on Machine Learning* (2016).
- [99] X. Chu et al. 2013. Discovering denial constraints. *VLDB Endowment* (2013).
- [100] Y. Chen et al. 2016. Ontological Pathfinding. In *International Conference on Management of Data*.
- [101] Y. Dong et al. 2023. DeepJoin: Joinable Table Discovery with Pre-trained Language Models. *Proc. VLDB Endow.* (2023).
- [102] Y. Huhtala et al. 1999. Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Computer Journal* (1999).
- [103] Y. Jin et al. 2020. Efficient Bidirectional Order Dependency Discovery. In *IEEE International Conference on Data Engineering (ICDE)*.
- [104] Y. Kaminsky et al. 2023. Discovering Similarity Inclusion Dependencies. *Proc. ACM Manag. Data* (2023).
- [105] Y. Zhang et al. 2020. Finding related tables in data lakes for interactive data science. In *ACM SIGMOD International Conference on Management of Data*.
- [106] Z. Wang et al. 2014. Knowledge graph embedding by translating on hyperplanes. *AAAI Conference on Artificial Intelligence* (2014).
- [107] Z. Wei et al. 2021. Algorithms for the discovery of embedded functional dependencies. *VLDB J.* (2021).
- [108] Z. Zhang et al. 2020. Learning Hierarchy-Aware Knowledge Graph Embeddings for Link Prediction. *AAAI Conference on Artificial Intelligence* (2020).
- [109] J. Euzenat. 2004. Ontology and Schema Evolution in Data Integration: Review and Assessment. In *On the Move to Meaningful Internet Systems*.
- [110] J. Fürnkranz. 1994. FOSSIL: A Robust Relational Learner.. In *European Conference on Machine Learning*.
- [111] P. KANELLAKIS. 1990. CHAPTER 17 - Elements of Relational Database Theory. In *Formal Models and Semantics*.
- [112] C. Marinescu. 2007. Discovering the Objectual Meaning of Foreign Key Constraints in Enterprise Applications. In *Working Conference on Reverse Engineering (WCRE 2007)*.
- [113] M. Spivey. 2008. A generalized recurrence for Bell numbers. *J. Integer Seq* (2008).
- [114] B. Thalheim. 2011. Recent Advances in Schema and Ontology Evolution. In *Schema Matching and Mapping*.

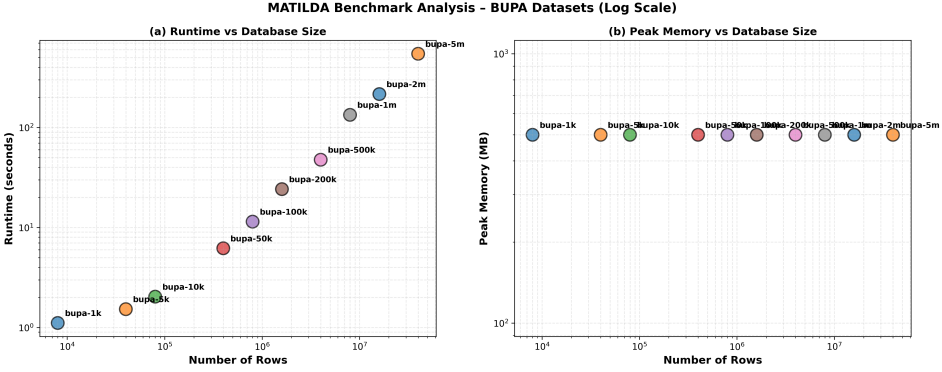


Fig. 4. **Scaling behavior on *Bupa-X* datasets (log scale).** We generate a family of scaled variants of *Bupa*, denoted *Bupa-X*, by uniformly duplicating existing tuples, so that all variants share the same schema and candidate rule space, and differ only in the number of rows. (a) Runtime as a function of the number of rows (log scale). Runtime increases with data volume, exhibiting a clear growth trend across scales. (b) Peak memory as a function of the number of rows (log scale).

## A Configuration of Competitors

We detail here the parameter configuration of Popper, AMIE, and Spider. For Popper, we use the default values `TIMEOUT = 600`, `EVAL_TIMEOUT = 0.001`, `MAX_LITERAL = 40`, `MAX_SOLUTIONS = 1`, `CLINGO_ARGS = ""`, `MAX_RULES = 2`, and `MAX_EXAMPLES = 10000`. To mine sufficiently many rules, we set `MAX_VARS = 6` and `MAX_BODY = 6`. For AMIE, we also use the default parameters JVM heap size limit = 15 GB, JAR file = `amie-milestone-intKB.jar`, minimum support threshold = 0%, minimum confidence threshold = 0%, minimum PCA confidence threshold = 0%, minimum head coverage threshold = 0%, minimum instantiation support threshold = 0%, and timeout = 300 (5 minutes). For Spider, we set `classpath = metanome-cli-1.2-SNAPSHOT.jar, SPIDER-1.2-SNAPSHOT.jar, -algorithm = de.metanome.algorithms.spider.SPIDERFile, -separator = ",", and -header`.

## B Additional Runtime Experiments

Figure 4 shows that peak memory remains essentially constant across scales, indicating that memory is dominated by schema-driven structures rather than by raw tuple count in this setting. Figure 5 shows that Memory usage increases with dataset volume but remains bounded and predictable, reflecting MATILDA’s controlled enumeration strategy.

*How to choose the rule length.* The larger the rule length  $l$  (parameter `max_length` in Algorithm 1), the longer the rule mining will take, and the longer the mined rules will be. One way to find the most useful  $l$  is to run MATILDA successively with larger  $l$ , starting from  $l = 2$ . As the runtime increases exponentially, the total loss of time when the right  $l^*$  is found is at most the runtime of  $l^*$ . However, previous work has suggested that rules with more than 3 atoms are rarely useful [59].

*How to choose the recursion depth.* The value of the recursion depth  $N$  (Definition 5.5) is naturally bound by the value of  $l$ . Since the results with a greater value of  $N$  will include the results with a smaller value of  $N$ , a simple strategy is to choose  $N = l$ , and to proceed as above for the search of the most useful  $l$ .

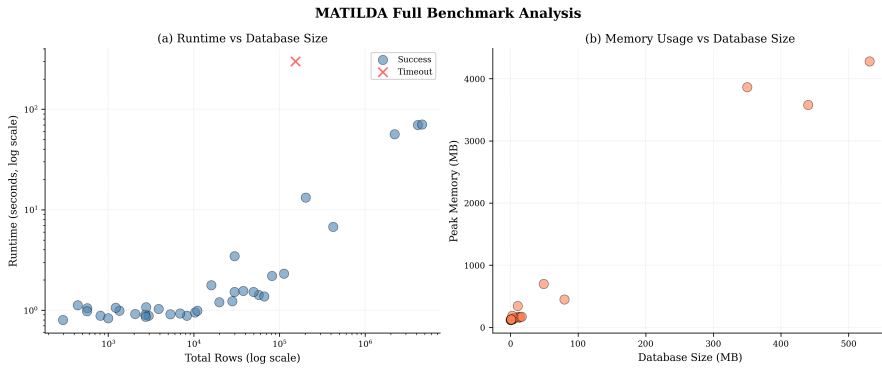


Fig. 5. Full benchmark analysis of MATILDA across heterogeneous databases. (a) Runtime as a function of the total number of rows (log-log scale). Each point corresponds to one database; successful runs are shown as circles, while timeouts are marked with crosses. Runtime grows smoothly with database size until a clear saturation regime. (b) Peak memory consumption as a function of database size.