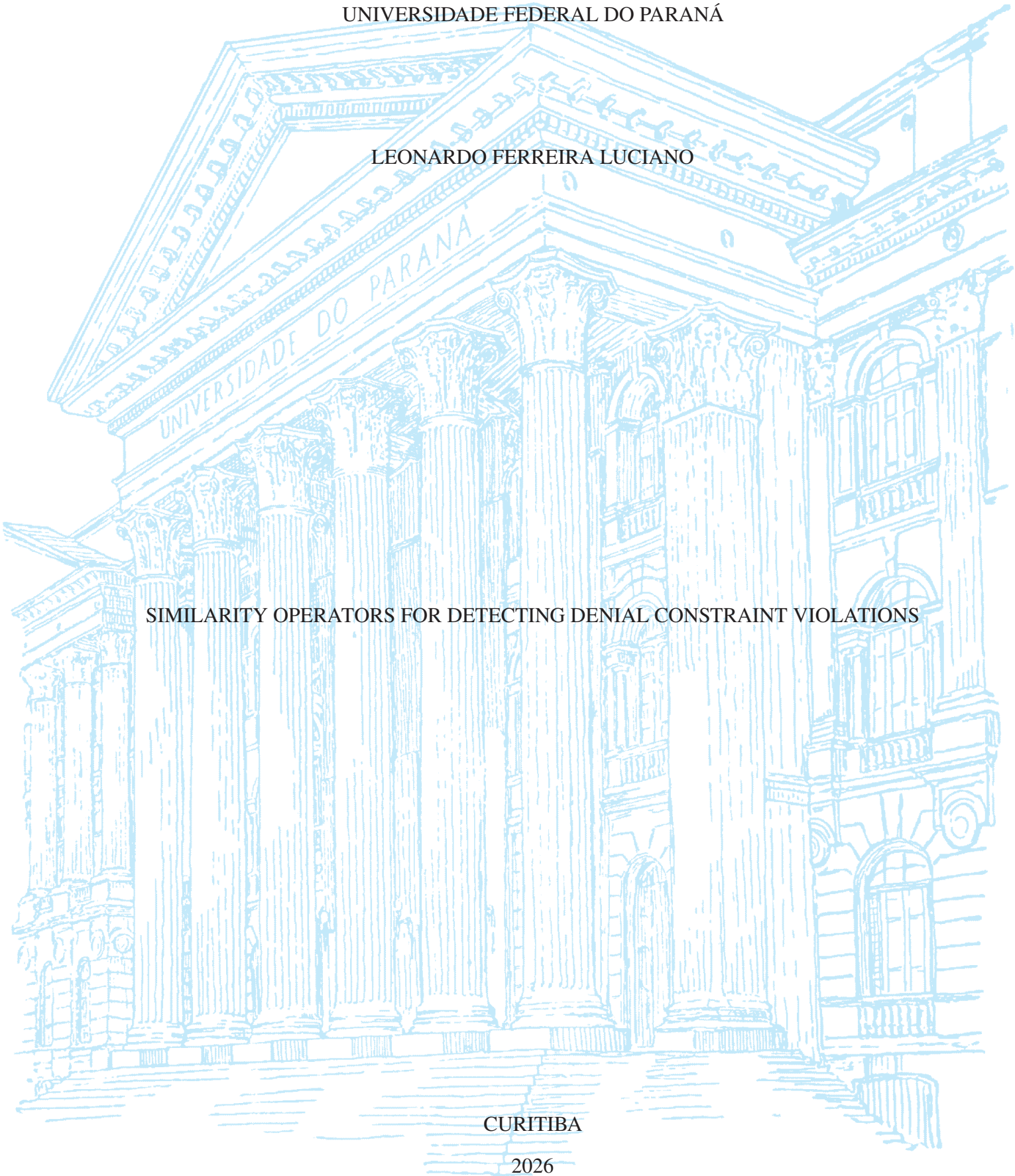
UNIVERSIDADE FEDERAL DO PARANÁ

LEONARDO FERREIRA LUCIANO



SIMILARITY OPERATORS FOR DETECTING DENIAL CONSTRAINT VIOLATIONS

CURITIBA

2026

LEONARDO FERREIRA LUCIANO

SIMILARITY OPERATORS FOR DETECTING DENIAL CONSTRAINT VIOLATIONS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Computação*.

Orientador: Eduardo Cunha de Almeida.

Coorientador: Eduardo Henrique Monteiro Pena.

CURITIBA

2026

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

Luciano, Leonardo Ferreira
Similarity operators for detecting denial constraint violations / Leonardo Ferreira Luciano. – Curitiba, 2026.
1 recurso on-line : PDF.

Dissertação (Mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.

Orientador: Eduardo Cunha de Almeida
Coorientador: Eduardo Henrique Monteiro Pena

1. Restrições (Inteligência artificial). 2. Semelhança (Filosofia). 3. Processamento eletrônico de dados. I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Almeida, Eduardo Cunha de. IV. Pena, Eduardo Henrique Monteiro. V. Título.

Bibliotecário: Douglas Lenon da Silva CRB-9/1892

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **LEONARDO FERREIRA LUCIANO**, intitulada: **Similarity Operators for Detecting Denial Constraint Violations**, sob orientação do Prof. Dr. EDUARDO CUNHA DE ALMEIDA, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 27 de Fevereiro de 2026.

Assinatura Eletrônica
02/03/2026 10:41:01.0
EDUARDO CUNHA DE ALMEIDA
Presidente da Banca Examinadora

Assinatura Eletrônica
03/03/2026 09:39:06.0
DANIEL KASTER
Avaliador Externo (UNIVERSIDADE ESTADUAL DE LONDRINA)

Assinatura Eletrônica
02/03/2026 10:47:34.0
SIMONE DOMINICO
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
09/03/2026 22:53:25.0
EDUARDO HENRIQUE MONTEIRO PENA
Coorientador(a)

RESUMO

A detecção de violações a restrições de integridade (ICs) é uma tarefa importante na limpeza de dados. Existem vários tipos de ICs, cada um deles expressando um conjunto diferente de restrições. Um tipo bem estudado de IC são as restrições de negação (DCs), que podem expressar um grande conjunto de restrições. Entretanto, DCs podem falhar em detectar violações quando na presença de dados textuais ruidosos, i.e., dados textuais contendo valores diferentes para representar a mesma entidade do mundo real. A existência de diferentes representações da mesma entidade pode ser causada, por exemplo, por erros de digitação, acrônimos, ou abreviações. Apesar da existência de vários sistemas para detectar violações de DCs, nenhum deles ataca o problema de detectar violações na presença de dados ruidosos. Além disso, outros tipos de ICs já foram propostos para lidar com dados ruidosos, mas nenhum deles pode expressar o mesmo conjunto de restrições representadas por DCs. Para atacar essas limitações das abordagens anteriores na detecção de violações a ICs, nós exploramos o uso de operadores de similaridade para detectar violações a DCs ao apresentar o *Similarity FAst Constraint-based Error DeTector* (`SimFACET`), um detector de violações a DCs com similaridade. Ele avalia eficientemente DCs contendo predicados com operadores de similaridade usando algoritmos exatos e aproximados para executar junções por similaridade. Para determinar a similaridade entre dados textuais, o detector suporta distância de edição entre dados textuais e distância de cossenos entre *text embeddings*. Nós também exploramos diferentes estratégias para o planejamento da order de processamento dos predicados de similaridade, tentando balancear custo de avaliação e a seletividade dos predicados para reduzir o tempo total de detecção. Para avaliar os algoritmos e estratégias propostos, nós conduzimos experimentos entre múltiplos *datasets* e utilizando DCs com diferentes características. Os resultados mostram que o `SimFACET` mantém alta acurácia enquanto consistentemente reduz o tempo de detecção, executando até 4× mais rápido do que versões de base.

Palavras-chave: Restrições de negação; similaridade; detecção de violações.

ABSTRACT

The detection of integrity constraints (ICs) violations is an important task in data cleaning. There are many types of ICs, each of them capable of expressing different sets of constraints. A well-studied type of IC are denial constraints (DCs), which can express a large set of constraints. However, DCs can fail in detecting violations when in the presence of noise string data, i.e., string data containing different values to represent the same real-world entity. The existence of these different representations of the same entity may be caused, for example, by typos, acronyms, or abbreviations. Despite the existence of several systems to detect DC violations, none of them addresses the problem of detecting violations in the presence of noisy data. Furthermore, other IC types have already been proposed to deal with noisy data, but none of them can express the same set of constraints represented by DCs. To address these limitations of the previous approaches in the detection of IC violations, we explore the use of similarity operators for detecting DC violations by presenting *Similarity FAst Constraint-based Error DeTector* (`SimFACET`), a violations detector for similarity DCs. It efficiently evaluates DCs containing predicates with similarity operators using exact and approximate algorithms for performing similarity joins. To determine string similarity, the detector supports edit distance between strings and cosine distance between text embeddings. We also explore different strategies for planning the processing order of the similarity predicates, trying to balance the evaluation cost and the selectivity of predicates to reduce the overall detection time. To evaluate the proposed algorithms and strategies, we have conducted experiments across multiple datasets and using DCs with different characteristics. The results show that `SimFACET` maintains high accuracy while consistently reducing detection time, executing up to 4× faster than baselines.

Keywords: Denial constraints; similarity; violations detection.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

CFD	Conditional functional dependency
DC	Denial constraint
DD	Differential dependency
FD	Functional dependency
HNSW	Hierarchical navigable small world
IC	Integrity constraint
IND	Inclusion dependency
IVF	Inverted file
LSH	Local-sensitive hashing
MD	Matching dependency
OD	Order dependency
SQL	Structured query language
UCC	Unique column combination
sIND	Similarity inclusion dependency

LIST OF SYMBOLS

φ	Denial constraint
τ	Similarity threshold
θ	Angle between two vectors

CONTENTS

1 INTRODUCTION

Integrity constraints (ICs) have been used in many data-related tasks, such as query optimization (Kossmann et al., 2021), inconsistency measurement (Livshits et al., 2021), and data cleaning (Abedjan et al., 2016). Enforcing a set of meaningful ICs increases data consistency and quality, enabling more reliable insights and better data-driven decisions in organizations. The IC-based data cleaning process can be divided into three main steps, which start by building the IC set to be enforced; followed by the detection of data violations of that IC set; and finishing with the repair of the errors pointed out by those data violations. In this work, we will focus on the second step, detecting data violations of a given IC set.

To illustrate the applicability of ICs in data quality, take as an example the relation `Employees` in Table 1.1, which stores employee records for a fictional company. Consider that this company has some employee-related policies, translated into the data rules over the `Employees` relation presented in Table 1.2. Rule (1) ensures a unique identification for each employee; rule (2) represents the geographical distribution of the employees over the offices of the company, one located in San Francisco and another in New York; and rule (3) represents a seniority rule. All three rules can be expressed by ICs, which can be used to detect and repair the erroneous data.

Table 1.1 - The `Employees` relation

	id	name	department	location	start_year	salary
t_1	101	Smith	Information Technology	San Francisco	2021	8000
t_2	102	Williams	Sales	New York	2022	9000
t_3	103	Jones	Sales	New York	2023	8000
t_4	104	Johnson	*IT*	San Francisco	2024	10000

Source: prepared by the author (2026)

Table 1.2 - Data rules for the `Employees` relation

(1)	Each employee must have a unique identifier
(2)	Employees from the same department must work at the same location
(3)	No employee must receive a salary lower than the salary of another employee who started after him in the same department

Source: prepared by the author (2026)

There are various types of ICs, each with different degrees of expressiveness, i.e., being capable of expressing different categories of data rules. Among these types, denial constraints (DCs) (Bertossi and Chomicki, 2004) are particularly interesting, as they generalize many other types of ICs, including unique column combinations (UCCs) (Heise et al., 2013); functional dependencies (FDs) (Wyss et al., 2001) and their conditional variation (CFDs) (Fan et al., 2008); and order dependencies (ODs) (Consonni et al., 2019). DCs can yet express some complex intra-relation constraints that cannot be represented by any of the mentioned constraint types. The high expressive power of DCs leads to the recent development of multiple systems for DC discovery (Bleifuß et al., 2017; Pena et al., 2022) and DC violations detection (Pena et al., 2021;

Luciano et al., 2023; Liu et al., 2024). Multiple errors repair systems also use DCs as their data rules language (Geerts et al., 2013; Rekatsinas et al., 2017). DCs have also been explored as an approach to solve real-world problems, as done by Tamalu et al. (2023).

If we consider only the exact-value comparison for the records, there are no violations of any of the three data rules in relation `Employees`. However, we should be more careful and consider the semantic value of data. In this case, we notice that the values of the `department` column in tuples t_1 (*Information Technology*) and t_4 (*IT*) refer to the same real-world entity, the Information Technology department. So, in the real world, the tuple pair (t_1, t_4) is a violation of rule (3), which has not been detected because of noisy data, in this case, multiple representations of the same department name caused by an abbreviation.

The described situations illustrate a limitation of traditional DCs, which is shared by the other types of ICs they generalize. Most works use DCs based on a fixed set of operators ($=, \neq, <, \leq, >, \geq$) and do not consider similarity-based operators (Chu et al., 2013; Bleifuß et al., 2017; Pena et al., 2021, 2022; Luciano et al., 2023; Liu et al., 2024). The works that define DCs with more flexible operators, allowing the notion of similarity, do not further explore the usage of similarity in operators (Bertossi, 2011; Rekatsinas et al., 2017; Heidari et al., 2019; Giannakopoulou et al., 2020). This absence of similarity-based operators has a special impact on DC violations detection when dealing with columns whose values are strings, as it is common for real data from string columns to be noisy, i.e., contain typos, abbreviations, acronyms, or any variations representing the same real-world entity. Therefore, considering similarity for string data in DCs can improve its usefulness in real datasets, given that it allows the detection of integrity violations even in the presence of noisy data.

However, the usefulness of including the notion of similarity in DCs is directly affected by the accuracy with which the similarity between two strings is measured. Many string similarity metrics have been proposed for this task. Metrics based on the number of editions that are needed to go from one string to another are known as edit distance metrics, with Hamming distance, Jaro-Winkler distance, and Levenshtein distance (Levenshtein, 1966) being examples of edit distance metrics. These metrics can capture only syntactic similarity, not considering semantic similarity directly. For example, edit distance with a reasonable threshold will be capable of capturing the similarity between strings *Information Technology* and *Informaton Technology*, but not between *Information Technology* and *IT*. With the use of text embeddings to represent strings (Mikolov et al., 2013; Devlin et al., 2019; Reimers and Gurevych, 2019), new metrics based on embedding similarity were proposed, including Euclidean distance and cosine distance. As embeddings include information about the context in the representation of each string, they can capture not only syntactic but also semantic similarity, being able to detect that the strings *Information Technology* and *IT* refer to the same department in the real world.

To address these questions, we present *Similarity FAst Constraint-based Error DeTector* (`SimFACET`), a system capable of detecting violations of DCs containing predicates with similarity operators. Similarity can be defined in multiple ways, and `SimFACET` supports two of the most commonly used options for strings and categorical data: edit distance between strings and cosine distance between text embeddings. Edit distance is faster but limited to syntactic similarity, while cosine distance increases detection time but can capture semantic similarity. To detect DC similarity violations across large datasets, we propose exact and approximate algorithms that leverage indexing techniques to evaluate the similarity operators efficiently. We integrate these algorithms into a robust detector of traditional DC violations and explore different strategies for planning the processing order of the similarity operators. In summary, our main contributions are:

- A system for detecting DC violations using similarity operators in large noisy datasets (Chapter 4);
- Exact and approximate algorithms that leverage indexing techniques to efficiently evaluate similarity operators, supporting both edit distance and cosine distance for text embeddings (Chapter 4);
- An experimental evaluation on multiple real and synthetic datasets, comparing detection time, accuracy of approximate approaches, and the impact of operator execution order (Chapter 5).

We provide background in Chapter 2, detail existing approaches to detect traditional DC violations in Chapter 3, discuss related work in Chapter 6, and summarize our conclusions in Chapter 7.

2 BACKGROUND AND PROBLEM STATEMENT

In this chapter, we detail the main concepts and notation used in this work. Section 2.1 presents the traditional DCs formalism, and defines the similarity operator and similarity DCs. In Section 2.2, we detail the similarity metrics explored in this work. Section 2.3 contains the formalization of our problem, presenting the research questions and stating the main hypothesis of this work. For an overview of the related work, refer to Chapter 6.

2.1 DENIAL CONSTRAINTS

Several types of integrity constraints (ICs) have been proposed, each of them capable of expressing constraints with different characteristics. The denial constraints (DCs) (Bertossi and Chomicki, 2004) are one of these IC types. In this work, we explore the inclusion of similarity operators in DCs to extend their expressive power, allowing for the accurate detection of violations in the presence of noisy data. DCs are chosen as our base formalism because of their already high expressive power. Any IC expressed as a unique column combination (UCC) (Heise et al., 2013); functional dependency (FD) (Wyss et al., 2001), and their conditional variation (CFD) (Fan et al., 2008); or order dependency (OD) (Consonni et al., 2019) can be represented as a DC. Also, some ICs cannot be represented by any of the previously mentioned formalisms but only by DCs.

DCs allow for the identification of conflicting combinations of column values. They define a set of predicates that cannot be true at the same time for any pair of distinct tuples from a relation. In this work, we follow Pena et al. (2021) and Liu et al. (2024), restricting our DCs to contain only predicates comparing different tuples, without allowing predicates comparing tuples with a constant. Let r be a relation with schema R and n tuples; A and B be columns of r; and t and t' be tuples in r, with $t \neq t'$. We denote $dom(A)$ the set of values of column A in table r; and $t.A$ the value of tuple t in column A. Given predicates $p_1, \cdots, p_m$ of the form $p : t.A \ f \ t'.B$, where $f : dom(A) \times dom(B) \rightarrow \{true, false\}$ is an operator, a DC φ is defined as follows:

$$\varphi : \forall t, t' \in r, \neg(p_1 \wedge \cdots \wedge p_m)$$

A tuple pair (t, t') satisfies a DC φ if it evaluates to true for φ. A violation of a DC φ is a tuple pair (t, t') that does not satisfy φ. We say that a DC φ holds on a relation r if it is not violated by any pair of distinct tuples in r. A relation r is inconsistent with respect to a DC φ if φ does not hold on r.

All the rules we have defined for the `Employees` relation, shown on Table 1.1, can be represented as DCs. For simplicity, the identifiers t and t' will be omitted from now on when defining DCs. Table 2.1 presents the DCs that express the data rules in Table 1.2. These DCs have the limitations discussed in Chapter 1, being unable to detect the real violations of the data rules in the `Employees` relation due to the noisy data it contains.

2.1.1 Similarity Operator

Given columns A and B, let function $d : dom(A) \times dom(B) \rightarrow \mathbb{R}$ be a similarity metric and $\tau_d \in \mathbb{R}$ be a similarity threshold. Given distinct tuples t and t', we define the similarity operator $\approx_{d,\tau_d}$: $dom(A) \times dom(B) \rightarrow \{true, false\}$ such that a predicate $p : t.A \approx_{d,\tau_d} t'.B$ evaluates to true if, and only if, $d(t.A, t'.B) \leq \tau_d$. Using an adequate metric and threshold, the similarity operator can detect both syntactic and semantic similarity between strings, evaluating to true

Table 2.1 - Traditional DCs for the `Employees` relation

(1)	$\varphi_1 : \neg(t.\texttt{id} = t'.\texttt{id})$
(2)	$\varphi_2 : \neg(t.\texttt{department} = t'.\texttt{department} \wedge t.\texttt{location} \neq t'.\texttt{location})$
(3)	$\varphi_3 : \neg(t.\texttt{department} = t'.\texttt{department} \wedge t.\texttt{start\_year} < t'.\texttt{start\_year} \wedge t.\texttt{salary} < t'.\texttt{salary})$

Source: prepared by the author (2026)

only when two strings, even if different, represent the same real-world entity. In this work, we propose implementations using edit distance and cosine distance as similarity metrics.

2.1.2 Similarity DCs

Traditionally, DCs are defined with $f \in \{=, \neq, <, \leq, >, \geq\}$ (Chu et al., 2013; Bleifuß et al., 2017; Liu et al., 2024). In this work, we extend this set of operators, supporting $f \in \{=, \neq, <, \leq, >, \geq, \approx_{d,\tau_d}\}$. We call similarity DCs those DCs containing at least one predicate using a similarity operator. With the similarity DCs, we can capture similarity between strings, allowing us to detect violations even in the presence of noisy textual data, expanding the expressive power of traditional DCs.

Back to the DCs defined on Table 2.1, we can now rewrite them as similarity DCs. The corresponding similarity DCs are shown in Table 2.2. Consider that the operator $\approx_{d,\tau_d}$ is perfect, i.e., evaluates to true if, and only if, the compared values represent the same entity in the real world. In this case, these DCs capture the data rules of the `Employees` table despite the noisy data. As the data rules (1) and (2), the DCs φ_1 and φ_2 have no violations. Furthermore, as rule (3), DC φ_3 is violated by the tuple pair (t_1, t_4), because $t_1.\texttt{department} \approx_{d,\tau_d} t_4.\texttt{department}$ (*Information Technology* $\approx_{d,\tau_d}$ *IT*); $t_1.\texttt{start\_year} < t_4.\texttt{start\_year}$ (2021 < 2024); and $t_1.\texttt{salary} < t_4.\texttt{salary}$ (8000 < 10000).

Table 2.2 - Similarity DCs for the `Employees` relation

(1)	$\varphi_1 : \neg(t.\texttt{id} = t'.\texttt{id})$
(2)	$\varphi_2 : \neg(t.\texttt{department} \approx_{d,\tau_d} t'.\texttt{department} \wedge t.\texttt{location} \neq t'.\texttt{location})$
(3)	$\varphi_3 : \neg(t.\texttt{department} \approx_{d,\tau_d} t'.\texttt{department} \wedge t.\texttt{start\_year} < t'.\texttt{start\_year} \wedge t.\texttt{salary} < t'.\texttt{salary})$

Source: prepared by the author (2026)

2.2 SIMILARITY METRICS

In this section, we present the two similarity metrics used in this work: edit distance between strings and cosine distance between text embeddings. These metrics are well-known similarity metrics. We define them as distances, so that we can define a threshold and use it in the similarity operator.

2.2.1 Edit Distance

Edit distance (Levenshtein, 1966) is defined as the minimum number of operations required to transform one string into another. An operation is a character insertion, deletion, or substitution. In this work, we follow the definition by Bocek et al. (2007). We refer to the edit distance between two strings s and s' as $ed(s, s')$. For example, taking the strings *Information Technology* and *IT* in Table 1.1, we have $ed(\textit{Information Technology}, \textit{IT}) = 20$. Given two strings s and s', the traditional dynamic programming algorithm for calculating the edit distance $ed(s, s')$ uses a matrix D of size $(|s| + 1) \cdot (|s'| + 1)$, where $i = 0, \cdots, |s|$ and $j = 0, \cdots, |s'|$, defined as follows:

$$
\begin{aligned}
&D[i, 0] = i,\\
&D[0, j] = j,\\
&D[i, j] = min(\\
&\qquad D[i-1, j] + 1\\
&\qquad D[i, j-1] + 1\\
&\qquad D[i-1, j-1] + (if\ s[i] = s'[j]\ then\ 0\ else\ 1)\\
&)
\end{aligned}
$$

A limitation of edit distance is that it can not be used to measure semantic similarity directly, but only syntactic similarity. It is not rare that strings with the same semantic value have very different syntax, i.e., that very different strings represent the same real-world entity. In these cases, to capture the similarity between the strings, the similarity threshold τ_{ed} will need to be too high, which can lead to other pairs of strings being wrongly considered similar. For example, to detect the similarity between the strings *Information Technology* and *IT*, we need $\tau_{ed} \geq 20$, as $ed(\textit{Information Technology}, \textit{IT}) = 20$. However, it will lead the strings *Information Technology* and *Sales* also being considered similar, as $ed(\textit{Information Technology}, \textit{Sales}) = 20$.

To exemplify the impact of that limitation in similarity DCs, let's return to the Employees relation on Table 1.1. Consider that the DC φ_3 employs edit distance as the metric of its similarity operator ($\approx_{ed,\tau_{ed}}$). If we use $\tau_{ed} < 20$, then we do not detect any violation and keep missing the detection of the violation (t_1, t_4). However, if we use $\tau_{ed} \geq 20$, then we detect the violation (t_1, t_4), but also incorrectly detect the violation (t_1, t_2), because of the incorrect evaluation of strings *Information Technology* and *Sales* as similar.

2.2.2 Cosine Distance

Text embeddings are high-dimensional dense vectors of numbers that can represent multiple types of data, including text, which is the focus of this work. Before embeddings, vectors of numbers are already used to represent text, with the index of each word in a dictionary being used as its representation, for example. One of the main innovations of embeddings is the inclusion of the concept of meaning in these representations. Text embedding models convert strings to embeddings using information about the context in which the strings are inserted. Some well-known embedding models are `Word2Vec` (Mikolov et al., 2013) and `BERT` (Devlin et al., 2019), which create embeddings from individual words; and `SBERT` (Reimers and Gurevych, 2019), which extends `BERT` to generate sentence embeddings more suitable for comparison.

Models can be trained specifically for the task for which they will be used, or they can be pre-trained on generic data and then used to solve general problems. Models trained specifically for a context are yet more powerful than pre-trained ones, being able to detect similarity in ways that are useful for their training context. In this work, we use only pre-trained models, as the generation of embeddings is not the focus of our research.

One of the similarity metrics that can be used to compare two embeddings is cosine similarity, which measures the similarity between two nonzero vectors using the cosine of the angle between them. Given two nonzero vectors $\mathbf{v}, \mathbf{v}' \in \mathbb{R}^l$ and θ being the angle between them, the cosine similarity between $\mathbf{v}$ and $\mathbf{v}'$ is:

$$cos(\theta) = \frac{\mathbf{v} \cdot \mathbf{v}'}{||\mathbf{v}|| \; ||\mathbf{v}'||}$$

The cosine distance between two embeddings $\mathbf{v}$ and $\mathbf{v}'$ is noted as $cd(\mathbf{v}, \mathbf{v}') = 1 - cos(\theta)$. Note that $\forall \mathbf{v}, \mathbf{v}', \; 0 \leq cd(\mathbf{v}, \mathbf{v}') \leq 2$. The closer to 0 the value of $cd(\mathbf{v}, \mathbf{v}')$ is, the more similar $\mathbf{v}$ and $\mathbf{v}'$ are. Threshold τ_{cd} for cosine similarity will depends of the embedding model used to generate the embeddings. Actually, for DCs employing cosine distance as metric for the similarity operator ($\approx_{cd,\tau_{dc}}$), the accuracy of the violations detection is highly tied to how well the embedding model captures the similarity in the context of the dataset where the DC is being evaluated.

2.3 RESEARCH QUESTIONS AND HYPOTHESIS

In this section, we present the research questions that have driven our work and state the hypothesis that comes from them.

To be useful for real-world applications, it is not enough for an IC to be able to detect a wide range of data integrity violations. It must be possible to detect these violations in a time small enough to make it feasible to solve real problems. In similarity DCs, the main challenge resides in the high complexity of evaluating similarity predicates. A naive algorithm will lead to a detection time that is prohibitive for real datasets. A common alternative to reduce the time required to solve this type of problem is indexing, which leads us to the Research Question 1.

Research Question 1 *Can indexing strategies significantly reduce the time required to detect violations of similarity denial constraints?*

Exact indexing strategies are generally enough for evaluating predicates employing operators using edit distance (Jiang et al., 2014), but for operators using cosine distance, most of the exact strategies fall short in significantly reducing the execution time because of the high dimensionality and density of text embeddings (Weber et al., 1998; Böhm et al., 2001; Jégou et al., 2011; Santana and Ribeiro, 2023). The solution for this limitation is the use of approximate indexes. A question that arises from the use of approximate indexes is about the accuracy of these strategies when compared with exact alternatives, which leads us to the Research Question 2.

Research Question 2 *Are approximate indexing strategies accurate for the detection of similarity denial constraints violations when compared with exact approaches?*

From these research questions, we formulate and now present a hypothesis that addresses these two questions at the same time. Our main objective in this work is to generate evidence of its truth. This hypothesis is stated in Hypothesis 1.

Hypothesis 1 *Exact and approximate indexing strategies significantly reduce the time required to detect violations of similarity denial constraints while keeping high accuracy when compared with exact approaches.*

2.4 CONCLUSION

In this chapter, we presented fundamental concepts for understanding this work. We provided the formal definition of DCs we adopt in this work, defined the similarity operators, and, from this definition, formalized the concept of similarity DCs. We also discussed how to measure similarity between strings, presenting definitions for edit distance between strings and cosine distance between text embeddings, which are the similarity metrics explored in this work. Finally, we presented our research questions and, derived from them, our hypothesis. In the next chapter, we present the framework for violations detection used in our solution.

3 DETECTING VIOLATIONS OF TRADITIONAL DCS

In this chapter, we present the general design of SimFACET. Several systems exist for detecting violations of traditional DCs (Pena et al., 2020, 2021; Liu et al., 2024). For SimFACET, we follow the design of FACET (Pena et al., 2021) due to its pipelined architecture, which enables natural integration of similarity operators. This architecture will be presented in Section 3.1, among other related concepts. For traditional operators, we reuse the algorithms of FACET and extend the framework with new algorithms for similarity-based predicates. The algorithms for traditional operators will be presented in Section 3.2. The algorithms for similarity-based operators will be discussed in Chapter 4.

3.1 REFINEMENTS PIPELINE

Refinements are special operators used to evaluate predicates. Their implementation varies depending on the operations used by the predicates they are designed to process. We now present refinements and the refinements pipeline, along with clusters and partitions, which are the structures used to store intermediates when processing the pipeline. These concepts are inspired by Bleifuß et al. (2017) and Pena et al. (2020, 2021).

3.1.1 Clusters and Cluster Pairs

Detecting violations of a DC in a relation requires evaluating pairs of distinct tuples of the relation for each predicate in the DC. Given a relation with n tuples, directly representing each tuple pair has space cost $O(n^2)$. Passing all these pairs through the pipeline requires large amounts of memory and leads to multiple function calls. To reduce the amount of data flowing through the pipeline, we use clusters and cluster pairs for a compact representation of tuple pairs. In the worst case, the space complexity of this approach is also $O(n^2)$, but it can be lower, reaching $O(n)$ in the best case. A cluster is a set of tuples in a table r. Given the clusters C_1 and C_2, a cluster pair (C_1, C_2) represents the set of all tuple pairs (t_1, t_2) such that $t_1 \in C_1$, $t_2 \in C_2$ and $t_1 \neq t_2$. Finally, given a cluster C_A, we denote as $dom(C_A)$ the set $\{t.A \in dom(A) \mid t \in C_A\}$ of values in column A of tuples t in C_A.

For example, the cluster pair $(\{t_1, t_3\}, \{t_1, t_2\})$ represents the set of tuple pairs $\{(t_1, t_2), (t_3, t_1), (t_3, t_2)\}$. Furthermore, we use the cluster C_r to represent the set of all tuples in a table r, with the set of all pairs of distinct tuples in r represented by the cluster pair (C_r, C_r). Finally, we call reflexive a pair of clusters C_1 and C_2 such that $C_1 = C_2$.

Consider the Employees relation shown in Table 1.1 and the DC φ_3 in Table 2.2. The set containing the $n \times (n-1)$ pairs of distinct tuples in the relation Employees is $\{(t_1, t_2), \cdots, (t_1, t_4), \cdots, (t_4, t_1), \cdots, (t_4, t_3)\}$. These pairs can be represented in space $O(n)$ by the cluster pair $(C_{Employees}, C_{Employees})$, where $C_{Employees} = \{t_1, \cdots, t_4\}$. The violation (t_1, t_4) of φ_3 in relation Employees can be represented by the cluster pair (C_1, C_2), where $C_1 = \{t_1\}$ and $C_2 = \{t_4\}$.

3.1.2 Partitions

Sometimes, when evaluating predicates or conjunctions of predicates, a single cluster pair is not enough to represent all the tuple pairs needed. For example, we can not use a single cluster pair to represent the set of tuple pairs $\{(t_1, t_2), (t_3, t_4)\}$. To keep the representation of tuple pairs compact

in such cases, we will use partitions. A partition is a set of cluster pairs that represents the union of all sets of tuple pairs represented by the cluster pairs in the partition. Formally, the partition $P = \{(C_1, C_2), \cdots, (C_{2m-1}, C_{2m})\}$ represents the set of tuple pairs $\bigcup_{(C_i,C_j)\in P}(C_i, C_j)$. Now, we can represent the tuple pairs in $\{(t_1, t_2), (t_3, t_4)\}$ using the partition $\{(\{t_1\}, \{t_2\}), (\{t_3\}, \{t_4\})\}$.

Note that different partitions can represent the same set of tuple pairs. Given the clusters $C_1 = \{t_1, t_3\}$, $C_2 = \{t_1, t_2\}$, $C_3 = \{t_1\}$ and $C_4 = \{t_4\}$, the partition $P = \{(C_1, C_2), (C_3, C_4)\}$ represents the set of tuple pairs $\{(t_1, t_2), (t_3, t_1), (t_3, t_2), (t_1, t_4)\}$. The same set of tuple pairs can be represented using the partition $P' = \{(C'_1, C'_2), (C'_3, C'_4)\}$, given the clusters $C'_1 = \{t_1\}$, $C'_2 = \{t_2, t_4\}$, $C'_3 = \{t_3\}$ and $C'_4 = \{t_1, t_2\}$.

Consider again the relation `Employees` shown in Table 1.1 and the DC φ_3 in Table 2.2. The set of all tuple pairs in the relation `Employees` can be represented by the partition $P_{Employees} = \{(C_{Employees}, C_{Employees})\}$, where $C_{Employees} = \{t_1, \cdots, t_4\}$. The violation (t_1, t_4) of φ_3 in relation `Employees` can be denoted as $P = \{(C_1, C_2)\}$, where $C_1 = \{t_1\}$ and $C_2 = \{t_4\}$.

3.1.3 Pipelined Evaluation

The refinement special operator receives a partition P and a predicate p, and returns a partition that represents the set of all tuple pairs represented by P that evaluate to true for the predicate p. For the relation `Employees` in Table 1.1, a refinement receiving the predicate (t.`department` $\approx_{d,\tau_d}$ t'.`department`) and the partition $\{(\{t_1, t_2\}, \{t_3, t_4\})\}$, representing the tuple pairs in $\{(t_1, t_3), (t_1, t_4), (t_2, t_3), (t_2, t_4)\}$, would return the partition $\{(\{t_1\}, \{t_4\}), (\{t_2\}, \{t_3\})\}$, representing the tuple pairs in $\{(t_1, t_4), (t_2, t_3)\}$.

To detect the violations of a DC $\varphi = \{p_1, \cdots, p_m\}$ in a relation r, we will use a refinements pipeline, which is an ordered tuple of refinements $(ref_1, \cdots, ref_m)$. We refer to the mapping from the predicates of φ to the refinements of the pipeline as refinements evaluation plan, which will be explored in Chapter 4. For now, without loss of generality, consider that the refinement plan is such that ref_i receives the predicate p_i, with $i = 1, \cdots, m$. The detection starts by passing the partition $P_r = \{(C_r, C_r)\}$ to refinement ref_1, which will return the partition P_1. Then the partition P_1 will be passed to refinement ref_2, which in turn will return the partition P_2. This process will continue until the refinement ref_m receives the partition P_{m-1} and returns the partition P_m. The partition P_m represents exactly the set of violations of DC φ in relation r.

Consider the relation `Employees` in Table 1.1 and the DC φ_3 in Table 2.2. Let p_1 : t.`department` $\approx_{d,\tau_d}$ t'.`department`, p_2 : t.`start_date` $<$ t'.`start_date` and p_3 : t.`salary` $<$ t'.`salary`. The refinements pipeline will be (ref_1, ref_2, ref_3) and consider that the refinement plan defines that the refinements ref_1, ref_2 and ref_3 will receive, respectively, the predicates p_1, p_2 and p_3. The detection starts with the refinement ref_1 receiving the partition $P_{Employees} = \{(\{t_1, t_2, t_3, t_4\}, \{t_1, t_2, t_3, t_4\})\}$ and returning the partition $P_1 = \{(\{t_1, t_4\}, \{t_1, t_4\}), (\{t_2, t_3\}, \{t_2, t_3\})\}$. Then, the refinement ref_2 will receive the partition P_1 and return the partition $P_2 = \{(\{t_1\}, \{t_4\}), (\{t_2\}, \{t_3\})\}$. Finally, the refinement ref_3 will receive the partition P_2 and return the partition $P_3 = \{(\{t_1\}, \{t_4\})\}$. The detection will then be finished, with the partition P_3 containing all violations of DC φ_3 in relation `Employees`.

3.2 ALGORITHMS FOR TRADITIONAL OPERATORS

When evaluating a DC, in addition to predicates using similarity operators, we also need to evaluate predicates using the traditional operators of DCs. We refer to traditional predicates as those that use equality, non-equality, or inequality operators. To process refinements receiving

traditional predicates, we employ the same algorithms used by the FACET (Pena et al., 2021), a traditional DC violations detector. We use the same pipelined architecture as FACET, so these algorithms can be easily integrated into our solution. This chapter briefly presents these algorithms. Section 3.2.1 presents algorithms for processing equality operators, Section 3.2.2 for non-equality operators, and Section 3.2.3 for inequality operators. The algorithms for processing refinements receiving predicates with similarity operators will be detailed in Chapter 4.

Following what is done by Pena et al. (2021), we divide predicates into classes based on the operators they use. This division enables the implementation of different algorithms to evaluate distinct classes of predicates, based on the characteristics of predicates in each class. Predicates using the equal operator ($=$) are equality predicates; predicates using the non-equal operator ($\neq$) are non-equality predicates; predicates using the less-than, less-equal-than, greater-than, or greater-equal-than operators ($<, \leq, >, \geq$) are inequality predicates, and predicates using the similar operator ($\approx_{d,\tau_d}$) are similarity predicates.

In addition to using different algorithms for different predicate classes, we will also adopt the same hybrid strategy used by Pena et al. (2021) to represent clusters. Bleifuß et al. (2017) uses arrays of integers to represent clusters, which is efficient when evaluating equality predicates, since equality algorithms only need to write and read data from clusters. But arrays of integers are not the best option for non-equality and inequality algorithms. These algorithms require many unions and differences of clusters, which are performed more efficiently when using compressed bitmaps instead of arrays of integers, as done by Pena et al. (2020). However, using compressed bitmaps introduces both computation and memory overheads that slow down equality algorithms. To avoid these overheads, we will use arrays of integers for equality algorithms and only employ compressed bitmaps for non-equality and inequality algorithms.

The combination of dynamic algorithms and data structures based on the classes of the predicates being processed allows the efficient detection of violations of DCs containing only traditional operators, as demonstrated by the results in Pena et al. (2021). Following this framework, we only need to focus on proposing efficient algorithms for processing refinements for similarity predicates to achieve efficient similarity DCs violation detection.

All algorithms are composed of a building phase and a probing phase. In the building phase, auxiliary data structures are created. These structures will be searched in the probing phase to generate the results. In addition, these algorithms allow the selection of only one of the two columns of the predicate to be used in the building phase. In these cases, the column with the least estimated cardinality will be used. The cardinality estimation will be performed using HyperLogLog (Flajolet et al., 2007).

For all algorithms, the input consists of a cluster pair (C_A, C_B) and a predicate p; and the output is a partition P' representing all the tuple pairs from (C_A, C_B) that evaluate to true for p. For simplicity, the algorithms are described processing a single input cluster pair (C_A, C_B), but, to process a refinement, we execute the described algorithm for each cluster pair in the received partition and then return the resultant partition, which is the union of all cluster pairs in the partitions returned by each execution.

3.2.1 Equalities

Predicates of the form $t.A = t'.B$ are equality predicates. The algorithm for processing refinements of equality predicates is based on the traditional hash join approach. Without loss of generality, assume that the column A will be used in the building phase. In this phase, we build a hash table H that maps each value $t.A$ of each tuple t in C_A to the cluster pair (C_1, C_2), where $C_1 = \{t' \in C_A \mid t'.A = t.A\}$ and $C_2 = \varnothing$. Then, in the probing phase, we iterate over the tuples in C_B. For each tuple t' in C_B, we retrieve from H the tuple pair (C_1, C_2) indexed by the value $t'.B$,

and add the tuple t' to cluster C_2. After iterating over all strings t', the probe phase is finished. Then, we initialize an empty partition P' to store the result. After, we iterate over the keys in H. For each key k, we get the cluster pair (C_1, C_2) indexed by k and add it to P'. After iterating over all keys k, we return the partition P and the algorithm terminates.

Consider the worst case where $|dom(C_A)| = |dom(C_B)| = |C_A| = |C_B| = n$. The cost of building the hash table H is $O(n)$, as we iterate over each one of the $|C_A|$ tuples in C_A. Consider that H is searched in $O(1)$. Then, the cost of the probing phase is also $O(n)$, as we iterate over the $|C_B|$ tuples in C_B. The cost of populating P' is $O(n)$ too, as we iterate over each one of the $|dom(C_A)|$ keys in H. Therefore, the cost of the equality algorithm is $O(n)$.

3.2.2 Non-Equalities

Predicates of the form $t.A \neq t'.B$ are non-equality predicates. The algorithm for processing refinements of non-equality predicates reuses most of the logic of the equality algorithm. We start by executing the building and probing phases of the equality algorithm, building the hash table H. After, we initialize an empty partition P' to store the result and iterate over the keys in H. For each key k in H, given the cluster pair (C_1, C_2) indexed by k in H, we add the cluster pair (C_1, C_2') to P', where $C_2' = C_B - C_2$. After iterating over all keys k, we return P', and the algorithm terminates.

Consider the worst case where $|dom(C_A)| = |dom(C_B)| = |C_A| = |C_B| = n$. The costs of building and probing phases are the same as in the equality algorithm, i.e., $O(n)$ for both phases. Let C_{diff} be the cost of calculating the set difference $C_B - C_2$, which varies depending on the data structure used in the clusters implementation. The cost of populating P' is $O(n \cdot C_{diff})$, as we need to compute a set difference with cost C_{diff} for each one of the $|dom(C_A)|$ keys in H. Therefore, the cost of the non-equality algorithm is $O(n \cdot C_{diff})$.

3.2.3 Inequalities

Predicates of the form $t.A\ f\ t'.B$, where $f \in \{<, \leq, >, \geq\}$, are inequality predicates. Other than equalities and non-equalities, `SimFACET` does not have only a single algorithm for processing refinements of inequality predicates. Instead, it selects the algorithm to be used to evaluate an inequality predicate based on characteristics of the input cluster pair and predicate. For traditional predicates, inequalities are the most expensive part of the refinements pipeline processing (Pena et al., 2021), which justifies the employment of three distinct strategies to process them.

The first inequality algorithm is `IEJoin`, which is proposed by Khayyat et al. (2015) and is used by Bleifuß et al. (2017) and Pena et al. (2021). `IEJoin` uses permutation arrays and offset arrays to efficiently evaluate two inequality predicates at one time. `IEJoin` is very fast, but it can only be used when there are pair of inequality predicates, and its performance degrades when evaluating low selectivity predicates. The second algorithm is `HSM`, which is used by Pena et al. (2020) and Pena et al. (2021). `HSM` is based on the traditional sort-merge join approach and can suffer performance issues in the presence of high cardinality columns. To address this problem, Pena et al. (2021) proposed the third algorithm, `BHSM`. It includes binning in `HSM` and, although outperformed by `HSM` in predicates with low cardinality columns, `BHSM` can scale better with the cardinality of the columns.

3.3 CONCLUSION

This chapter presented the framework used by `SimFACET` for detecting DC violations. We started by describing the refinements pipeline, which allows for the easy inclusion of algorithms

for evaluating predicates with similarity operators. Then, clusters and partitions have been presented, as they are crucial for the efficient representation and processing of tuple pairs in the pipeline. Last, we presented the algorithms used for processing traditional operators, along with the data structures used to represent the intermediates in those algorithms. In the next chapter, we present the approaches employed in `SimFACET` to process refinements of predicates using similarity operators.

4 DETECTING VIOLATIONS OF SIMILARITY DCS

In this chapter, we present the algorithms used by SimFACET for evaluating predicates using the similarity operator ($\approx_{d,\tau_d}$). These algorithms are the core of the refinements of similarity DCs. Similarity operators can be implemented using any similarity metric (d). In this work, we explore two of them, proposing one algorithm for edit distance between strings and three algorithms for cosine distance between text embeddings. Before presenting the algorithms, we discuss the relation between evaluating a similarity operator and performing a similarity join in Section 4.1. Then, Section 4.2 describes our algorithm for edit distance, and Section 4.3 presents the cosine distance algorithms.

All the algorithms use the same input and output types. They receive a cluster pair (C_A, C_B) representing the tuple pairs to be evaluated on predicate $p : t.A \approx_{d,\tau_d} t'.B$ using the similarity metric d and threshold τ_d, and return a partition P' representing all the tuple pairs that evaluate true to p given d and τ_d. For simplicity of exposition, we present these algorithms with a single cluster pair (C_A, C_B) as input. In practice, they are executed for each cluster pair in a refinement input partition P. Furthermore, the similarity indexes are always built using the combination of a column and a cluster with the smaller cardinality. However, for easy presentation, we convey that the indexes are built using column A and cluster C_A.

4.1 SIMILARITY JOIN

Let V and V' be sets containing values of any type. We define the similarity join $V \bowtie_{d,\tau_d} V'$ as the problem of finding all value pairs (v, v') where $v \in V$, $v' \in V'$, and v is similar to v' for the metric d and threshold τ_d. Given a cluster pair (C_A, C_B) and a similarity predicate $p : t.A \approx_{d,\tau_d} t'.B$, note that the problem of evaluating the tuple pairs from (C_A, C_B) for p can be handled as a $V \bowtie_{d,\tau_d} V'$ similarity join problem. First, we extract all values $t.A$ from tuples t in C_A, creating a set $V = \bigcup_{t \in C_A} t.A$ and mapping each value $v \in V$ to the set of tuples $\{t \in C_A \mid t.A = v\}$. We do the same for C_B and V'. Now we perform the similarity join $V \bowtie_{d,\tau_d} V'$ and, for each resulting value pair (v, v'), we use the generated mappings to build cluster pairs (C'_A, C'_B). These resulting cluster pairs represent the tuple pairs from (C_A, C_B) that evaluate to true for the predicate p.

Let's consider the worst case, where $|dom(C_A)| = |dom(C_B)| = |C_A| = |C_B| = n$. The generation from cluster C_A of value set V has cost $O(n)$. Generating V' from C_B has a similar cost. So, the total cost before the join is $O(n)$. After the join, the cluster pairs (C'_A, C'_B) are generated, for which the cost depends on the number of joined value pairs (v, v'). If all evaluated value pairs are similar, the cost is $O(|dom(C_A)| \cdot |dom(C_B)|) = O(n^2)$. However, it is not common in real-world scenarios that all evaluated pairs are similar, so, in most cases, the cost will be lower than $O(n^2)$. The join itself, when implemented in a naive way, i.e., using nested loops, always has a cost of $O(n^2)$. Therefore, the similarity join is the most time-expensive step in the similarity refinements processing. For this reason, our work focuses on reducing the join time by employing indexing strategies.

4.2 EDIT DISTANCE ALGORITHM

In this section, we present our implementation of similarity operators using edit distance ($\approx_{ed,\tau_{ed}}$) in SimFACET. For edit distance, it was possible to, using exact indexing, achieve an execution

time that is reasonable for real-world applications. So, we do not explore approximate indexing for edit distance, implementing a single algorithm: `EDFastSS`.

4.2.1 EDFastSS

The `EDFastSS` algorithm is based on the `FastSSwC` index presented by Bocek et al. (2007), which is a time-efficient approach to solve the string similarity search problem, which consists of, given a string s and a string set S', finding all strings $s' \in S'$ similar to s. Next, we first present `FastSSwC` and then detail our algorithm.

`FastSSwC` employs character deletions on strings to generate their deletion neighborhoods. Given a string s, a character deletion $del(s, e) = s_v$ is the operation of generating a variation string s_v by deleting the character at position e of s. For example, consider the string `Sales`: $del(\texttt{Sales}, 0) = \texttt{ales}$, $del(\texttt{Sales}, 1) = \texttt{Sles}$, and so on. The k-deletion neighborhood of s is the set of all strings that can be obtained recursively by deleting at most k characters from s. Formally, the k-deletion neighborhood of a string s is:

$$U_{del}(s, k) = \bigcup_{e=0}^{|s|-1} (\{del(s, e)\} \cup U_{del}(del(s, e), k - 1))$$

Following the previous example, the string $del(\texttt{Sales}, 0) = \texttt{ales}$ is in $U_{del}(\texttt{Sales}, 2)$, as well as the string $del(\texttt{ales}, 0) = \texttt{les}$, among all the other strings that can be generated from the string `Sales` by deleting at most 2 characters.

An important property of character deletion neighborhoods is that, given strings s and s', and a threshold τ_{ed}, if $ed(s, s') \leq \tau_{ed}$, then $U_{del}(s, \tau_{ed}) \cap U_{del}(s', \tau_{ed}) \neq \varnothing$. This property is the main idea behind `FastSSwC`, assuring the completeness of its result. It allows for the pruning of several strings from the set where the search is being performed, reducing the number of candidates whose edit distance needs to be calculated.

Given a string s, a string set S', and a threshold τ_{ed}, `FastSSwC` efficiently finds all strings $s' \in S'$ such that $ed(s, s') \leq \tau_{ed}$ by building an index that maps each deletion neighbor s'_v of each string s' in S' to the set $\{c \in S' \mid s'_v \in U_{del}(c, \tau_{ed})\}$. `FastSSwC` then searches this index for each delete neighbor s_v of s, performing the union of the sets found. The result of these unions is the candidate string set $S_c = \{c \in S' \mid U_{del}(c, \tau_{ed}) \cap U_{del}(s, \tau_{ed}) \neq \varnothing\}$. For each string s, the edit distance between s and each string c in the candidate set S_c is obtained using the traditional dynamic programming algorithm for calculating Levenshtein distance, adding to the result set the strings c similar to s. At the end of execution, the result is the set $\{s' \in S' \mid ed(s, s') \leq \tau_{ed}\}$.

`EDFastSS` is presented in Algorithm 1. First, we initialize an empty partition P', which will store the final result. After, we build a hash table H_A that maps the string value $t.A$ of each tuple t in C_A to all the tuples t' in C_A such that $t'.A = t.A$. Similarly, we build a hash table H_B for column B and cluster C_B. Then, using the keys from H_A and the threshold τ_{ed}, we build a `FastSSwC` index I. Next, we iterate over the keys from H_B. For each string s_B from H_B, we create a cluster C'_B with the tuples indexed in H_B by s_B, and initialize an empty cluster C'_A. From index I, we retrieve all candidates with a non-empty deletion neighborhood intersection with s_B, filtering from them the strings s_A such that $ed(s_B, s_A) \leq \tau_{ed}$. For each filtered string s_A, we add the tuples indexed in H_A by s_A to the cluster C'_A. After iterating over all strings s_A, we add the cluster pair (C'_A, C'_B) to the partition P'. After iterating over all strings s_B, we return the partition P' and `EDFastSS` terminates.

Let's consider the worst case for the algorithm, where $|dom(C_A)| = |dom(C_B)| = |C_A| = |C_B| = n$. The cost of building the hash table H_A is $O(n)$, as we iterate over each one of the $|C_A|$ tuples in C_A. Similarly, the cost of building H_B is $O(n)$. Consider that H_A and H_B are

Algorithm 1 `EDFastSS`$((C_A, C_B), p, \tau_{ed})$

```
1:  P' ← ∅
2:  H_A ← CreateValuesIndex(C_A, A)
3:  H_B ← CreateValuesIndex(C_B, B)
4:  I ← FastSSwCIndex(H_A.keys(), τ_ed)
5:  for all s_B ∈ H_B.keys() do
6:      C'_B ← H_B.get(s_B)
7:      C'_A ← ∅
8:      S_c ← FastSSwCSearch(I, s_B)
9:      for all s_A ∈ S_c do
10:         if ed(s_B, s_A) ≤ τ_ed then
11:             C'_A ← C'_A ∪ H_A.get(s_A)
12:         end if
13:     end for
14:     P' ← P' ∪ {(C'_A, C'_B)}
15: end for
16: return P'
```

both searched in $O(1)$. Let l be the average string size in $dom(C_A)$. The cost of building the `FastSSwC` index I is $O(n \cdot l^{\tau_{ed}})$, as `FastSSwC` generates a deletion neighborhood for each string in $dom(C_A)$ and as each string neighborhood generation costs $O(l^{\tau_{ed}})$. Consider that the index I is searched in $O(1)$. So, the cost of the `FastSSwC` search is the same as generating a deletion neighborhood, $O(l^{\tau_{ed}})$. The cost of the dynamic programming algorithm for calculating the Levenshtein distance is $O(l^2)$. Therefore, the cost of `EDFastSS` is $O(n \cdot ((l^{\tau_{ed}}) + (|S_c| \cdot l^2)))$, indicating the algorithm is sensitive both to the string length and the number of candidates selected.

4.3 COSINE DISTANCE ALGORITMS

In this section, we present our implementations of similarity operators using cosine distance ($\approx_{cd,\tau_{cd}}$) in `SimFACET`. For cosine distance, exact approaches, including tree-based ones, have a low or no impact on efficiency, because embeddings are vectors both dense (Santana and Ribeiro, 2023) and have high dimension (Weber et al., 1998; Böhm et al., 2001; Jégou et al., 2011). So, to achieve a detection that is time feasible for real-world applications, we explore approximate indexing approaches. We propose two approximate algorithms for cosine distance: `CDAllIV` and `CDSampledIVF`. To evaluate the accuracy of the approximate algorithms, we also implement an exact baseline algorithm: `CDFlat`.

4.3.1 CDFlat

The `CDFlat` algorithm is our exact baseline using nested loops. It is shown in Algorithm 2. First, we initialize an empty partition P', which will store the final result. After, we build a hash table H_A that maps the string value $t.A$ of each tuple t in C_A to all the tuples t' in C_A such that $t'.A = t.A$. Similarly, we build a hash table H_B for column B and cluster C_B. Next, we iterate over the keys from H_B. For each embedding $\mathbf{v_B}$ from H_B, we create a cluster C'_B with the tuples indexed in H_B by $\mathbf{v_B}$, and initialize an empty cluster C'_A. Then, from the keys from H_A, we filter the embeddings $\mathbf{v_A}$ such that $cd(\mathbf{v_B}, \mathbf{v_A}) \leq \tau_{cd}$. For each filtered embedding $\mathbf{v_A}$, we add the tuples indexed in H_A by $\mathbf{v_A}$ to the cluster C'_A. After iterating over all embeddings $\mathbf{v_A}$, we add the

cluster pair (C'_A, C'_B) to the partition P'. After iterating over all embeddings $\mathbf{v_B}$, we return the partition P' and `CDFlat` terminates.

Algorithm 2 `CDFlat`$((C_A, C_B), p, \tau_{cd})$

```
1:  P' ← ∅
2:  H_A ← CreateValuesIndex(C_A, A)
3:  H_B ← CreateValuesIndex(C_B, B)
4:  for all v_B ∈ H_B.keys() do
5:      C'_B ← H_B.get(v_B)
6:      C'_A ← ∅
7:      for all v_A ∈ H_A.keys() do
8:          if cd(v_B, v_A) ≤ τ_cd then
9:              C'_A ← C'_A ∪ H_A.get(v_A)
10:         end if
11:     end for
12:     P' ← P' ∪ {(C'_A, C'_B)}
13: end for
14: return P'
```

Let's consider the worst case for the algorithm, where $|dom(C_A)| = |dom(C_B)| = |C_A| = |C_B| = n$. The cost of building the hash table H_A is $O(n)$, as we iterate over each one of the $|C_A|$ tuples in C_A. Similarly, the cost of building H_B is $O(n)$. Consider that H_A and H_B are both searched in $O(1)$. Therefore, the cost of `CDFlat` is $O(n^2)$, as for each vector in $dom(C_B)$, it iterates over each vector in $dom(C_A)$.

4.3.2 CDAllIVF

The CDAllIVF algorithm relies on inverted file (IVF) indexes, more specifically, `IVFFlat` (Sivic and Zisserman, 2003; Facebook, 2025), as it achieves higher accuracy than existing alternatives, as `IVFADC` (Jégou et al., 2011; Yang et al., 2020). The idea behind using an IVF index is to create lists of embeddings indexed by a key embedding in a way that, if a query embedding is similar to a key embedding, then it is similar to the embeddings in the list indexed by it. So, we only need to calculate the distance between the query embedding and the key embeddings, and search for embeddings similar to the query embedding only within the lists indexed by the most similar key embeddings. To create those lists, we cluster the embeddings using `k-means` (Smile, 2025). The `k-means` clusters will be our lists, and their centroids will be our key embeddings. For the number of lists used in our implementation, we chose a value commonly used as a starting point for IVF indexes: the square root of the number of embeddings being indexed (Facebook, 2025).

The `k-means` algorithm receives a set of vectors V and an integer c, and clusterizes the vectors from V in c clusters. It starts by randomly selecting c vectors from V, referred to as centroids, and associates each one of them with a cluster. Then, it will iterate over each vector v in V and include v in the cluster associated with the centroid most similar to v. After, for each cluster, the associated centroid will be updated, receiving the value of the mean vector of all vectors in that cluster. These two last steps, attributing vectors in V to clusters and updating the centroids, will be repeated until the difference between the old centroids and the new ones is smaller than a defined threshold, or until the max number of iterations is reached. `SimFACET` uses 10^{-4} as the `k-means` threshold and limits the number of iterations to 20.

`CDAllIVF` is presented on Algorithm 3. First, we initialize an empty partition P', which will store the final result. After, we build a hash table H_A that maps the string value $t.A$ of each tuple t in C_A to all the tuples t' in C_A such that $t'.A = t.A$. Similarly, we build a hash table H_B for column B and cluster C_B. After, we use all keys from H_A to create an IVF index I. Then, we iterate over the keys from table H_B. For each embedding $\mathbf{v_B}$ from H_B, we create a cluster C'_B with the tuples indexed in H_B by $\mathbf{v_B}$, and initialize an empty cluster C'_A. Then, we calculate the cosine distance between $\mathbf{v_b}$ and each key in I, order the keys based on smaller distance, and get the lists L indexed by 1% of the keys from I most similar to $\mathbf{v_b}$. We iterate over each list L and filter the embeddings $\mathbf{v_A}$ from L such that $cd(\mathbf{v_B}, \mathbf{v_A}) \leq \tau_{cd}$. For each filtered embedding $\mathbf{v_A}$, we add the tuples indexed in H_A by $\mathbf{v_A}$ to the cluster C'_A. After iterating over all lists L, we add the cluster pair (C'_A, C'_B) to the partition P'. After iterating over all embeddings $\mathbf{v_B}$, we return the partition P' and `CDAllIVF` terminates.

Algorithm 3 `CDAllIVF`$((C_A, C_B), p, \tau_{cd})$

```
1:  P' ← ∅
2:  valuesIndexA ← CreateValuesIndex(C_A, A)
3:  valuesIndexB ← CreateValuesIndex(C_B, B)
4:  indexA ← BuildIVFIndex(valuesIndexA.keys())
5:  for all v_B ∈ valuesIndexB.keys() do
6:      C'_B ← valuesIndexB.get(v_B)
7:      C'_A ← ∅
8:      for all L ∈ SearchIVFIndex(indexA, v_B) do
9:          for all v_A ∈ L do
10:             if cd(v_B, v_A) ≤ τ_cd then
11:                 C'_A ← C'_A ∪ valuesIndexA.get(v_A)
12:             end if
13:         end for
14:     end for
15:     P' ← P' ∪ {(C'_A, C'_B)}
16: end for
17: return P'
```

Let's consider the worst case for the algorithm, where $|dom(C_A)| = |dom(C_B)| = |C_A| = |C_B| = n$. The cost of building the hash table H_A is $O(n)$, as we iterate over each one of the $|C_A|$ tuples in C_A. Similarly, the cost of building H_B is $O(n)$. Consider that H_A and H_B are both searched in $O(1)$. The cost of building the IVF index is $O(n \cdot \sqrt{n})$, as it is the cost of the `k-means` algorithm using $\sqrt{n}$ clusters and limited to 20 iterations. Consider that the average cluster size is $n/\sqrt{n}$. The cost of searching the IVF index is also $O(n^2)$, as for each vector in $dom(C_B)$, we search $0.01 \cdot \sqrt{n}$ lists and verify the $n/\sqrt{n} =$ vectors in each of those lists. Therefore, the cost of `CDAllIVF` is $O(n^2)$, but it generally executes faster than `CDFlat` by checking only a fraction of the lists.

4.3.3 CDSampledIVF

The `CDSampledIVF` algorithm is a variation of `CDAllIVF`. It is shown in Algorithm 4. In `CDAllIVF`, we execute `k-means` with all the key embeddings from H_A. It helps to create an accurate IVF index, i.e., an IVF index that, given an embedding $\mathbf{v}$, has lists L where the average distance between $\mathbf{v}$ and the embeddings in L is highly proportional to the distance between $\mathbf{v}$ and the indexing key of L. However, the use of all embeddings also increases the execution time

of k-means and, therefore, elevates the IVF index building time. CDSampledIVF addresses this question by using only 10% of the embeddings in the k-means execution. The used embeddings are randomly selected from all keys from H_A. After the clusterization, each of the embeddings $\mathbf{v}$ that was not used in k-means is added to the list with the key most similar to $\mathbf{v}$. It reduces the accuracy of the IVF index, but also significantly reduces the building time of the index. To compensate for the reduction in the index accuracy, we verify more lists in CDSampledIVF. While in CDAllIVF we verify only 1% of the lists, in CDSampledIVF we increase this fraction to 10%.

Algorithm 4 CDSampledIVF$((C_A, C_B), p, \tau_{cd})$

```
1:  P' ← {}
2:  valuesIndexA ← CreateValuesIndex(C_A, A)
3:  valuesIndexB ← CreateValuesIndex(C_B, B)
4:  sampledEmbeddings ← SampleEmbeddings(valuesIndexA.keys())
5:  indexA ← BuildIVFIndex(valuesIndexA.keys(), sampledEmbeddings)
6:  for all v_B ∈ valuesIndexB.keys() do
7:      C'_B ← valuesIndexB.get(v_B)
8:      C'_A ← {}
9:      for all L ∈ SearchIVFIndex(indexA, v_B) do
10:         for all v_A ∈ L do
11:             if cd(v_B, v_A) ≤ τ_cd then
12:                 C'_A ← C'_A ∪ valuesIndexA.get(v_A)
13:             end if
14:         end for
15:     end for
16:     P' ← P' ∪ {(C'_A, C'_B)}
17: end for
18: return P'
```

4.4 INTEGRATING SIMILARITY OPERATORS INTO THE REFINEMENTS PIPELINE

While refinements can be evaluated in any order, we strategically order them in the pipeline based on predicate selectivity—the proportion of tuple pairs a predicate filters out. Predicates that evaluate to true for fewer tuple pairs have higher selectivity. We order refinements with traditional operators from most to least selective, following (Pena et al., 2021): equalities (=), inequalities ($<, \leq, >, \geq$), and non-equalities ($\neq$).

Given the column A, consider that $freq(v)$ denotes the frequency of a value v in $dom(A)$. Given an equality predicate p of the form $p : t.A = t'.A$, the number of tuple pairs selected by p is $\sum_{v \in dom(A)} freq(v)^2$. Now consider a non-equality predicate p' of the form $p' : t.A \neq t'.A$. The tuple pairs selected by p' are the set of all tuple pairs filtered out by p, so the number of tuple pairs filtered in by p' is $\sum_{v \in dom(A)} freq(v) \cdot (n - freq(v))$. Therefore, as we can intuitively see that equalities have higher selectivity than non-equalities, the defined planning ordering allows for fewer intermediates to be processed in the pipeline, which is expected to reduce detection time.

Predicates using similarity operators ($\approx_{d,\tau_d}$) are expected to have selectivity only slightly lower than equalities. Therefore, because equalities are highly selective, similarities are also expected to be highly selective. However, since the indexes employed in similarity algorithms are

more complex than those used for other operators, processing similarity refinements is expected to be slower than other refinements. To understand the impact of these two factors, we explore three distinct strategies for planning the pipeline with similarity refinements. The first strategy, labeled “I”, prioritizes reducing the number of intermediates by applying similarity refinements immediately after equalities. The second, named “C”, focuses on delaying the cost of similarity refinements as much as possible, placing them at the end of the pipeline, after non-equalities. Finally, the third strategy, referenced “B”, represents a balance between the other two, with similarity refinements placed right after inequalities but before non-equalities.

4.5 CONCLUSION

In this chapter, we discussed how to process the similarity operators and presented the implementations used in `SimFACET`. We started discussing the relation between evaluating a similarity predicate and processing a similarity join. Then, the algorithms used for processing similarity refinements have been described, detailing the exact indexing strategy for edit distance, and the approximate indexing and sampling strategies for cosine distance. Finally, we discussed how to plan the evaluation of similarity predicates in the refinements pipeline, focusing on balancing the cost of processing those predicates and their selectivity. The next chapter presents our experimental evaluation and discusses the results obtained.

5 EXPERIMENTAL EVALUATION

This chapter presents our experimental evaluation of SimFACET. We start by describing the experimental settings in Section 5.1, including the datasets used, the DCs associated with each dataset, the host machine specifications, and other details of our experimental protocol. After, we present and discuss the results of our experiments. Section 5.2 focuses on the detection time of SimFACET using different similarity algorithms and planning strategies, showing how these results help us to answer the Research Question 1. In Section 5.3, we focus on Research Question 2, investigate the impact of the approximate approaches employed in CDAllIVF and CDSampledIVF algorithms on the accuracy of violations detection, comparing them with the exact results obtained with CDFlat.

5.1 EXPERIMENTAL SETTINGS

The datasets used in our experimental evaluation are presented in Table 5.1. Along with the dataset names and numbers of rows, we present the column cardinalities for each column used in a DC, as it affects the complexity of the algorithms by determining the number of distinct values being processed. It also contains the column values' average lengths, i.e., the average number of characters in the column values, which impact the complexity of EDFastSS. The similarity DCs used for each of the datasets are shown in Table 5.2. These DCs allow us to see the behavior of SimFACET when processing similarity predicates among equality, non-equality, and inequality predicates.

The Tax and Flights datasets are used by Pena et al. (2021). Tax is a syntactic compilation of tax-related records of US individuals generated using the data generator presented by Fan et al. (2008); and Flights contains records from domestic US flights. They have columns with well-controlled data and are common in the evaluation of systems for detecting DC violations. The Inspections dataset (ChicagoDataPortal, 2025) contains records from inspections of food establishments in the city of Chicago and is available on the city's open data portal. The Contacts dataset (NYCOpenData, 2025b) contains records derived from multiple dwelling registration forms of the city of New York and is available on the city's open data portal. The Jobs dataset (NYCOpenData, 2025a) contains records of data derived from job application forms submitted in the city of New York and is also available on the city's open data portal. The Inspections, Contacts, and Jobs datasets have columns containing noise data, which makes them useful to evaluate the efficiency of our system in the presence of real-world noisy data.

We ran all experiments on a server with an AMD EPYC 7401 virtualized with KVM (2.0 GHz, eight cores, eight threads, 512 KiB of private L1d cache, 512 KiB of private L1i cache, 4 MiB of private L2 cache, 128 MiB of private L3 cache); 32 GB of RAM; 1 TB of SDD; Debian GNU/Linux 12; OpenJDK 64-Bit Server VM 17.0.14+7-Debian-1deb12u1 with the JVM head size set to 16 GB. We terminated any execution exceeding one hour. Each reported time is the average of six executions. For the embeddings-based algorithms, all the string values have been previously converted to embedding values using the embedding model paraphrase-mpnet-base-v2 (HuggingFace, 2025). As any embeddings-based algorithm requires creating embeddings from strings, and as we focus on the impact of our algorithms on similarity join execution, we do not report the time required to create embeddings in our

Table 5.1 - Datasets used in our experiments.

Dataset	Number of rows	Column	Col. cardinality	Col. values avg. length
Flights	1M	custom_origin	42k	28
		custom_destination	19k	25
		distance	2k	-
Tax	1M	custom_city	30k	20
		salary	10k	-
		rate	851	-
Inspections	286k	aka_name	32k	19
		dba_name	33k	-
Contacts	782k	business_city	5k	10
		business_street_name	49k	12
		business_house_number	22k	-
		type	9	-
		first_name	50k	7
		last_name	71k	8
Jobs	27k	job_description	22k	135
		other_description	870	11
		initial_cost	4k	-
		total_est_fee	5k	-

Source: prepared by the author (2026)

experiments. Unlike the time spent creating embeddings, the time required to build the indexes is a central aspect of our analysis; therefore, we present it for all algorithms.

5.2 DETECTION TIME

In this section, we present our evaluation of SimFACET in terms of detection time. We investigate the impact of each implementation of the similarity operator as well as the impact of each of the pipeline planning strategies. Furthermore, the effect of variations in the similarity threshold is investigated for both edit distance and cosine distance similarity metrics.

5.2.1 Algorithms and Planning Strategies

Figure 5.1 shows the detection time for each combination of algorithm and refinement pipeline planning strategy.

For edit-distance similarity, the results show the impact of string length on EDFastSS detection time. No matter the pipeline planning strategy, EDFastSS did not finish the detection for Jobs(φ_5) because of the high average length of values in column job_description. The string's length is also contributing to the timeout of EDFastSS in Flights(φ_1) and Inspections(φ_3) when similarities are processed after the non-equalities, as they generate a large amount of cluster pairs for the similarity refinements, forcing the building of many small indexes with relatively long strings. The timeout in Tax(φ_2) had a different origin, occurring because of the long processing time of the pair of inequalities.

Table 5.2 - Similarity DCs used in our experiments.

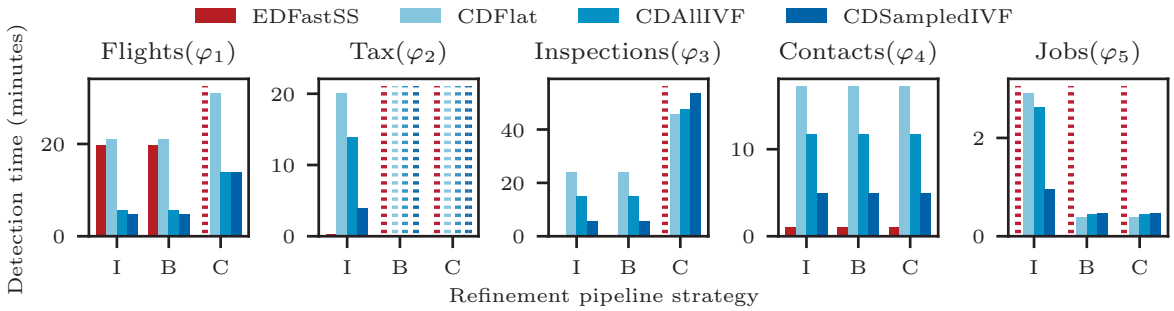
DC num.	Dataset	Denial constraint
φ_1	Flights	$\neg(t.\texttt{custom\_origin} \approx_{d,\tau_d} t'.\texttt{custom\_destination}$ $\wedge\, t.\texttt{custom\_destination} \approx_{d,\tau_d} t'.\texttt{custom\_origin}$ $\wedge\, t.\texttt{distance} \neq t'.\texttt{distance})$
φ_2	Tax	$\neg(t.\texttt{custom\_city} \approx_{d,\tau_d} t'.\texttt{custom\_city}$ $\wedge\, t.\texttt{salary} > t'.\texttt{salary}$ $\wedge\, t.\texttt{rate} < t'.\texttt{rate})$
φ_3	Inspec.	$\neg(t.\texttt{aka\_name} \approx_{d,\tau_d} t'.\texttt{aka\_name}$ $\wedge\, t.\texttt{dba\_name} \neq t'.\texttt{dba\_name})$
φ_4	Contacts	$\neg(t.\texttt{business\_city} \approx_{d,\tau_d} t'.\texttt{business\_city}$ $\wedge\, t.\texttt{business\_street\_name} \approx_{d,\tau_d} t'.\texttt{business\_street\_name}$ $\wedge\, t.\texttt{business\_house\_number} = t'.\texttt{business\_house\_number}$ $\wedge\, t.\texttt{type} = t'.\texttt{type}$ $\wedge\, t.\texttt{first\_name} \approx_{d,\tau_d} t'.\texttt{first\_name}$ $\wedge\, t.\texttt{last\_name} \approx_{d,\tau_d} t'.\texttt{last\_name})$
φ_5	Jobs	$\neg(t.\texttt{job\_description} \approx_{d,\tau_d} t'.\texttt{job\_description}$ $\wedge\, t.\texttt{other\_description} \approx_{d,\tau_d} t'.\texttt{other\_description}$ $\wedge\, t.\texttt{initial\_cost} < t'.\texttt{initial\_cost}$ $\wedge\, t.\texttt{total\_est\_fee} > t'.\texttt{total\_est\_fee})$

Source: prepared by the author (2026)

For cosine distance similarity, the results show that, in general, indexing with `CDAllIVF` reduces the detection time compared to `CDFlat`, with speedups of 1.9× for “I”, 1.7× for “B”, and 1.3× for “C”. Using sampling with `CDSampledIVF` improves performance even more than `CDAllIVF`, with speedups of 4.0× for “I”, 2.8× for “B”, and 1.7× for “C”. Two exceptional cases occur in `Inspections`(φ_3) and `Jobs`(φ_5) when the similarities are processed later in the pipeline, where the detection time increases with the use of indexing and sampling. In both cases, the increase in detection time is caused by a large number of cluster pairs containing small cardinality clusters being generated by the previous operators. The time cost of indexing and sampling does not pay off for these small cardinality clusters, making `CDFlat` the better option in this situation. As occurs for edit distance, for strategies "B" and "C", none of the algorithms for cosine distance finish their detection within the time limit for `Tax`(φ_2) because of the cost of processing the inequalities.

For both edit distance and cosine distance similarities, the results indicate that it is better to process the similarity predicates earlier in the pipeline, reducing the total number of tuple pairs being evaluated and consequently shortening the detection time. The exceptional result in `Jobs`(φ_5), where the detection time is short when processing the cosine distance similarities later, can be explained by the low selectivity of the similarity predicates in DC φ_5 for the dataset `Jobs`, which contains many similar values in the columns `job_description` and `other_description`. Given the low selectivity of those predicates, reducing the number of intermediates in evaluating similarities earlier in the pipeline does not compensate for the time cost of processing the similarity operators with a larger number of tuple pairs.

Figure 5.1 - Detection time of similarity algorithms for each refinement pipeline planning strategy. The dashed lines represent cases that exceed the execution time limit of one hour. For `EDFastSS`, the τ_{ed} threshold used was 2. For `CDFlat`, `CDAllIVF` and `CDSampledIVF`, the τ_{cd} threshold used was 0.15.

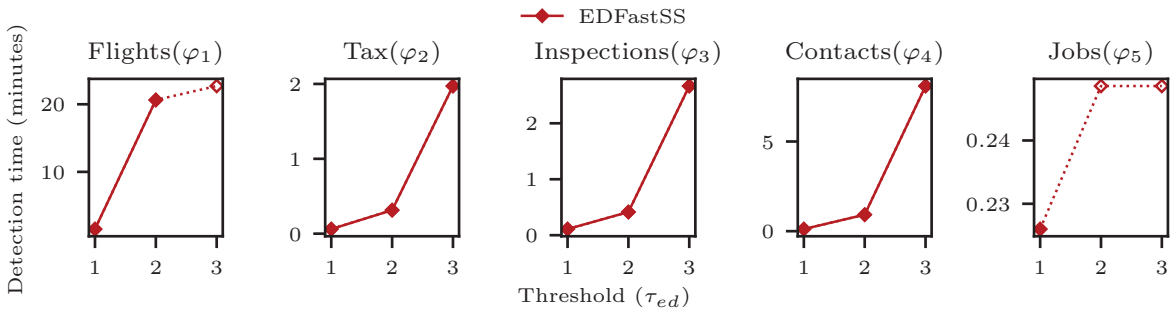


Source: prepared by the author (2026)

5.2.2 Similarity Thresholds

Figure 5.2 shows the detection times for the edit distance algorithm when varying the τ_{ed} threshold from 1 to 3. The results indicate that the time required for detection increases rapidly as the threshold increases. This is an expected result, as the threshold is directly related to the size and build time of the `FastSSwC` index. For datasets with larger strings, the detection time rapidly exceeds the one hour limit, as occurs for `Flights`(φ_1) with $\tau_{ed} = 3$, and for `Jobs`(φ_5) with $\tau_{ed} = 2$ and $\tau_{ed} = 3$.

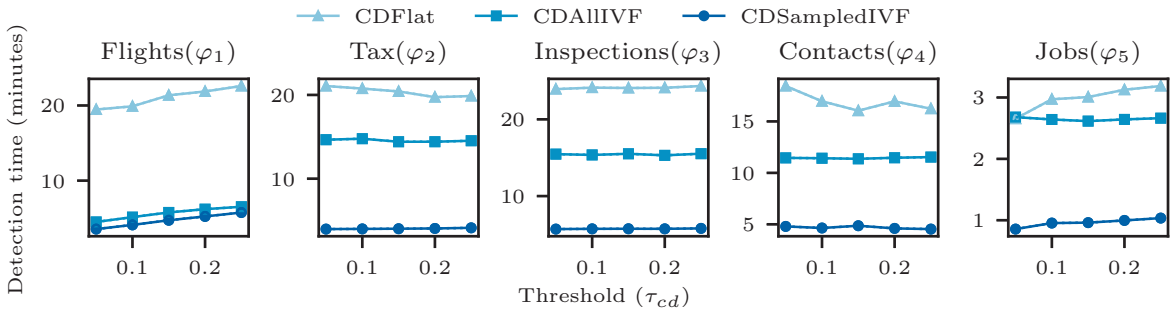
Figure 5.2 - Detection time of similarity algorithm based on edit distance when varying the τ_{ed} threshold. The dashed lines and unfilled markers represent cases that exceed the execution time limit of one hour. The refinement pipeline strategy adopted was "I".



Source: prepared by the author (2026)

Figure 5.3 shows the detection times of the cosine distance algorithms when varying the τ_{cd} threshold from 0.05 to 0.25. As expected, the results indicate that increasing the threshold has a low impact on the algorithms' detection time, as it has no influence on the size or build time of the IVF index, and only slightly affects the number of intermediates passed to the next operators in the refinement pipeline. Besides, those results demonstrate that the improvements from using indexing and sampling shown in Figure 5.1 are consistent across multiple thresholds, reinforcing the contribution of the `CDSampledIVF` algorithm to improving detection performance.

Figure 5.3 - Detection time of similarity algorithms based on cosine distance when varying the τ_{cd} threshold. The dashed lines and unfilled markers represent cases that exceed the execution time limit of one hour. The refinement pipeline strategy adopted was "I".



Source: prepared by the author (2026)

5.2.3 Concluding Remarks

The results show that the proposed indexing strategies can significantly reduce the detection time of similarity DC violations, answering the Research Question 1. For cosine distance, they show that indexing is capable of reducing detection time, which is potentiated by the use of sampling in `CDSampledIVF`. The combination of these two strategies in `CDSampledIVF` has been up 4× faster than `CDFlat`, the baseline algorithm without indexing. For edit distance, the use of indexes allows for the quick detection of similarity DC violations, requiring only a fraction of the time needed for detection using cosine distance. Finally, the results indicate that processing similarity predicates early in the pipeline positively impacts detection efficiency, as, because of the high selectivity of these predicates, it reduces the cost of evaluating the following predicates by reducing the total number of intermediates being processed.

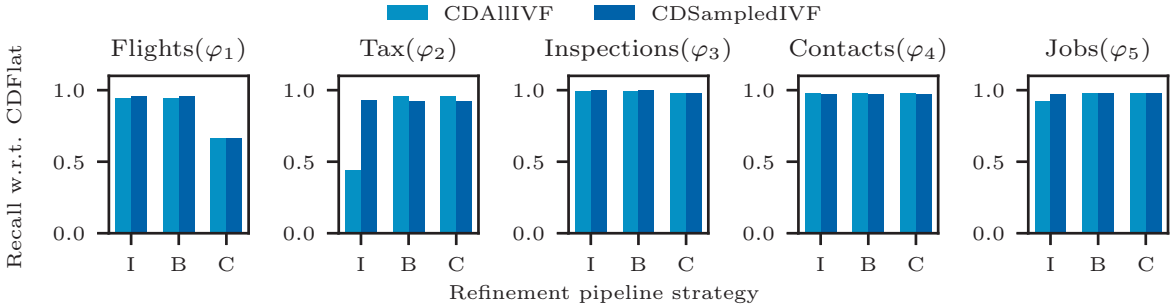
5.3 APPROXIMATE INDEXES ACCURACY

This section presents our investigation into the accuracy of the approximate indexing strategies used in `SimFACET` for speeding up the detection of similarity DCs using cosine distance. We compare the violations detected using the approximate algorithms with the violations detected using the `CDFlat` exact algorithm. We explore the effect of varying the pipeline planning strategies and the similarity thresholds for our approximate algorithms. The results are shown in terms of recall and F1 score. The precision is not presented as it is always 1 for any scenario.

5.3.1 Algorithms and Planning Strategies

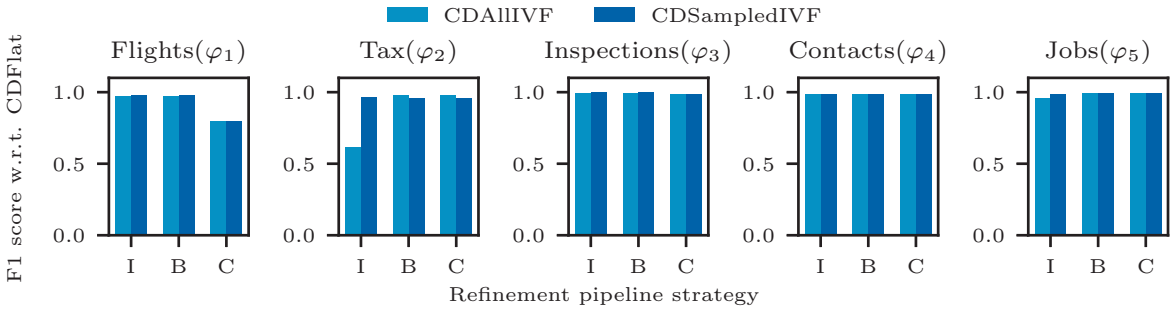
Figure 5.4 and Figure 5.5 present, respectively, the recall and the F1 score with respect to the exact similarity algorithm `CDFlat` of the approximate similarity algorithms `CDAllIVF` and `CDSampledIVF`. The results show that `CDAllIVF` and `CDSampledIVF` algorithms allow for a reliable detection in almost any case. Furthermore, the sampling approach used in `CDSampledIVF` slightly improves detection compared to `CDAllIVF`. The reason is that, as it verifies a higher number of lists to compensate for the decrease in clustering quality, the probability of a similar embedding not being verified is lower in `CDSampledIVF` when compared with `CDAllIVF`.

Figure 5.4 - Recall with respect to `CDFlat` of approximate similarity algorithms based on cosine distance for each refinement pipeline planning strategy. The recall was calculated over detected violations. The τ_{cd} threshold used was 0.15.



Source: prepared by the author (2026)

Figure 5.5 - F1 score with respect to `CDFlat` of approximate similarity algorithms based on cosine distance for each refinement pipeline planning strategy. The F1 score was calculated over detected violations. The τ_{cd} threshold used was 0.15.



Source: prepared by the author (2026)

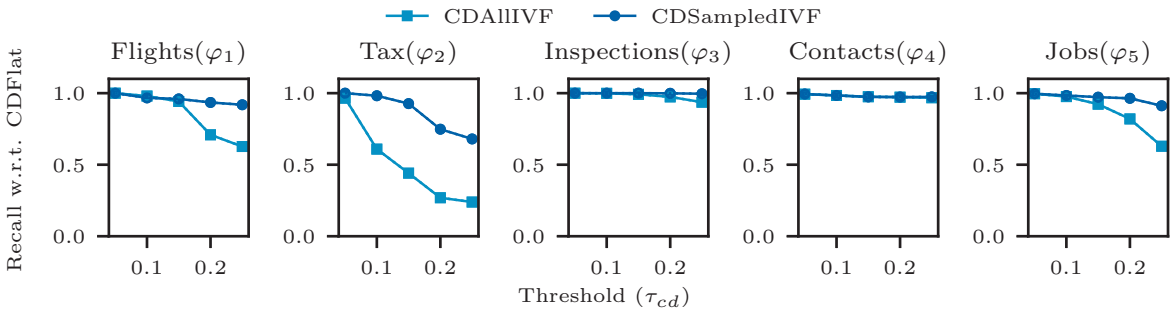
5.3.2 Similarity Thresholds

Figure 5.6 and Figure 5.7 show, respectively, the recall and the F1 score with respect to the exact similarity algorithm `CDFlat` of the approximate similarity algorithms `CDAllIVF` and `CDSampledIVF` when varying the τ_{cd} threshold from 0.05 to 0.25. The results indicate that the detection reliability is negatively affected by increasing the similarity threshold. The reason is that there are more embeddings in lists that are not being verified because their key embeddings are not so similar to the query embedding. Here we see again that the sampling approach from `CDSampledIVF` improves the detection, making it more reliable as the threshold increases.

5.3.3 Concluding Remarks

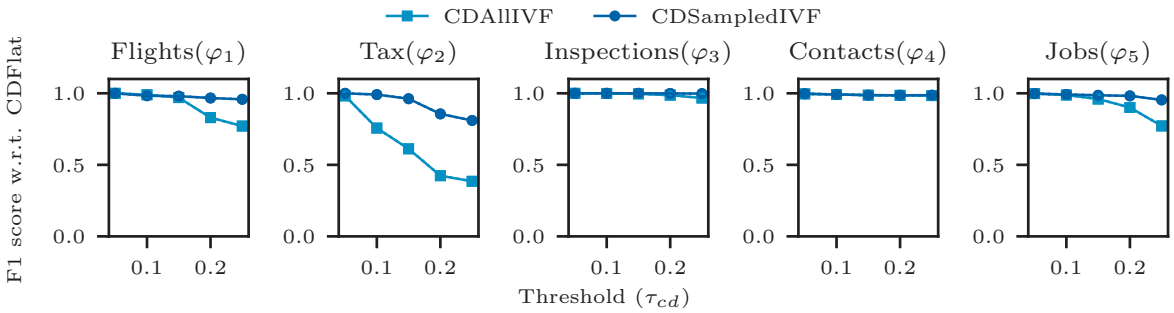
The results show that the proposed approximate indexing strategies allow for the highly accurate detection of similarity DC violations, answering the Research Question 2. The indexing in `CDAllIVF` provides good detection accuracy, even using generic IVF parameters without tuning. Furthermore, the sampling strategy in `CDSampledIVF` improves the indexing accuracy even more by allowing the verification of a larger number of embeddings. Sampling also makes the detection more stable as the similarity thresholds grow.

Figure 5.6 - Recall with respect to `CDFlat` of approximate similarity algorithms based on cosine distance when varying the τ_{cd} threshold. The recall was calculated over detected violations. The refinement pipeline strategy adopted was "I".



Source: prepared by the author (2026)

Figure 5.7 - F1 score with respect to `CDFlat` of approximate similarity algorithms based on cosine distance when varying the τ_{cd} threshold. The F1 score was calculated over detected violations. The refinement pipeline strategy adopted was "I".



Source: prepared by the author (2026)

5.4 CONCLUSION

This chapter described our experimental evaluation of `SimFACET` and discussed its results. The datasets and the associated similarity DCs were presented, along with the experimental setup and protocol details. Then, we discussed the results obtained with our experiments, focusing first on detection time and then on the accuracy of the approximate indexes. The experiments were designed and carried out with the main objective of understanding the impact of the approximate indexing approaches used in the cosine distance algorithms, of the pipeline placement strategies for the similarity operators, and of the variations on the similarity threshold in use. We also answered our two research questions, presenting evidence that supports our hypothesis. The next chapter provides an analysis of related works, comparing the existing literature with this work.

6 RELATED WORK

This chapter presents works associated with our research. In Section 6.1, we present an overview of the IC-based data cleaning area, with special attention to DCs. We focus on violations detection, but also discuss other steps in data cleaning, aiming to localize our work in the broader area of data cleaning. Then, in Section 6.2, we provide an overview of other integrity constraints, including those that handle the notion of similarity between strings, showing their limitations when compared with similarity DCs. Finally, in Section 6.3, we review works associated with strategies for performing similarity joins, showing the reasons that sustain the choices made in our algorithms.

6.1 DATA CLEANING

The IC-based data cleaning process can be divided into three main steps: constraints discovery, violations detection, and error repair. Several works have been proposed to deal with each one of these steps. Multiple systems have been proposed to automatically discover constraints, using both DCs (Chu et al., 2013; Bleifuß et al., 2017; Pena and de Almeida, 2018; Pena et al., 2019, 2022; Bian et al., 2024) and other ICs (Huhtala et al., 1999; Wyss et al., 2001; Fan et al., 2008; Song and Chen, 2011; Schirmer et al., 2020; Kaminsky et al., 2023; Kuang et al., 2024). The detection of violations has also been a topic of interest in the data cleaning area, with works exploring automated detection of violations of DCs (Heidari et al., 2019; Pena et al., 2020, 2021; Luciano et al., 2023; Liu et al., 2024) and other ICs (Fan et al., 2008). Finally, systems have also been proposed for automated error repair (Geerts et al., 2013; Rekatsinas et al., 2017; Mahdavi and Abedjan, 2020).

6.1.1 Constraints Discovery

Defining constraints for which a dataset must be consistent helps to improve the quality of the data and, consequently, of the insights organizations can extract from it. The constraints discovery step focuses exactly on this problem, consisting of defining the set of ICs that a dataset must be consistent with, i.e., the set of ICs that must not be violated in the dataset. Domain specialists can manually define the set of IC that a dataset must respect, but it is a costly and error-prone process. The alternative is the use of automated tools for discovering ICs.

Chu et al. (2013) provide a formal definition for the problem of discovering DCs from data and propose `FASTDC`, the first algorithm for automated DCs discovery. It is capable of discovering both exact and approximate DCs, i.e., DCs that are violated by a limited quantity of tuple pairs in the dataset. `Hydra` (Bleifuß et al., 2017) is a DC discovery system faster than `FASTDC`, but limited to exact DCs. While `FASTDC` requires the processing of all tuple pairs in the input dataset, `Hydra` employs strategies to reduce the number of tuple pairs evaluated. `BFASTDC` (Pena and de Almeida, 2018) also does not require the processing of all tuple pairs, just as `Hydra`, but can discover approximate DCs, similarly to `FASTDC`. `DCFinder` (Pena et al., 2019) improves `BFASTDC` by using predicate selectivity for reducing the number of operations performed. The strategies of `DCFinder` are reused in a new framework for DC violations discovery proposed by Pena et al. (2022). There is also another line of DCs discovery tools based on learning strategies, for which we can cite `DCMiner` (Bian et al., 2024), an approach based on reinforcement-learning, as an example.

Although SimFACET is a violations detector and not a constraints discoverer, it reuses and adapts some strategies applied to discoverers, given that evaluating predicates is necessary in both these categories of systems. The array of integers representation of clusters used in SimFACET is the same as that employed by Hydra. The idea of deciding the order in which evaluate the predicates based on their classes, considering the expected selectivity for each one of them, is derived from BFASTDC.

6.1.2 Violations Detection

After defining, manually or automatically, the set of ICs that must be respected by a dataset, it is important to have an efficient tool for verifying whether the data is consistent with the defined ICs. This step, referred to as violations detection, is the focus of our work, as SimFACET is a violations detector. We reuse the ideas proposed in other works about DC violations detection, adapting them for the inclusion of predicates with similarity operators.

Given that many ICs classes can be translated into SQL queries, a natural approach to detect violations is executing SQL queries over a DBMS. This approach can also be used with DCs, as they can be easily translated into SQL queries (Chu et al., 2013). However, this SQL-based approach can lead to complex self-joins, where the number of intermediates grows rapidly, making the time to detect all violations prohibitive. To address this problem, specific-purpose systems to detect DC violations have been proposed (Pena et al., 2020, 2021; Liu et al., 2024).

One of the first specific purpose system to detect DC violations was VioFinder (Pena et al., 2020). Using information about predicate selectivity, it creates a predicate evaluation pipeline where only the tuple pairs that satisfy the previous predicates are evaluated against the current predicate. VioFinder also uses partitions to store tuple pairs in a more efficient way than the exhaustive enumeration of tuple pairs. It adopts and improves some strategies used in the DCs discovery tool Hydra (Bleifuß et al., 2017), as in its discovery process, Hydra has to detect violations of candidate DCs. VioFinder is already capable of detecting DC violations faster than SQL-based approaches, but is improved by the techniques proposed in Pena et al. (2021), where FACET is presented. FACET applies hybrid data structures and algorithms to store and evaluate tuple pairs. Luciano et al. (2023) extend FACET by proposing projection algorithms to allow visualization of detected violations. Recently, Rapidash (Liu et al., 2024) was proposed, being able to detect DC violations even faster than FACET. None of these detectors considers similarity operators in DCs, supporting only the traditional operators.

The pipeline architecture of SimFACET, including the use of compact data structures to represent tuple pairs, is based on the design of VioFinder and FACET. The compressed bitmaps, used in SimFACET as one option to represent intermediates, are from VioFinder. The use of hybrid data structures to represent intermediates and of different algorithms depending on the class of the predicate being processed are employed on FACET. The algorithms for processing traditional predicates are also the same used by FACET.

6.1.3 Error Repair

After detecting violations of the rules defined for a dataset, the natural next step is think about how to repair the errors pointed out by them, making the dataset consistent with the ICs that represent the rules for the data. Besides IC violations, other sources can indicate errors in data, such as external data sources or statistical analysis. Some error repair systems integrate multiple sources to find and repair data errors.

Llunatic (Geerts et al., 2013) is an error repair system based on constraints, using SQL queries to detect violations and proposing strategies to repair them. HoloClean (Rekatsinas et al., 2017) uses multiple sources to find errors in data, including DC violations. HoloClean does not focus on detecting violations, but expects a detector capable of pointing out the violations. Knowing the existing violations, it unifies other error signs and uses a probabilistic model to repair data.

As the detection of IC violations is part of the error repairing process, the improvement in detection represented by the inclusion of similarity operators in DCs can positively impact the quality of the error repair from tools like Llunatic and HoloClean, or any other error repairing tool that employs DC violations detection for mapping the errors to be repaired. With violations more faithful to the real world, the quality of the repairs performed by those tools will increase.

6.2 SIMILARITY DCS COMPARED WITH OTHER ICS

Traditional DCs have the capacity of expressing all the rules expressed by multiple other IC types, as unique column combination (UCC) (Heise et al., 2013); functional dependency (FD) (Wyss et al., 2001), and their conditional variation (CFD) (Fan et al., 2008); and order dependency (OD) (Consonni et al., 2019). Similarity DCs extend the expressiveness of traditional DCs by including the notion of string similarity, improving the range of rules that can be expressed using DCs. The idea of including the similarity notion in ICs to improve their expressive power is not new. Some other works already proposed the relaxation of existing formalisms to accommodate similarity (Fan, 2008; Song and Chen, 2011; Schirmer et al., 2020; Kaminsky et al., 2023; Kuang et al., 2024), but none of them can express the same set of violations represented by similarity DCs.

An example of similarity inclusion in ICs is matching dependencies (MDs). They are a generalization of functional dependencies (FDs) that allow relaxation of equality constraints to similarity ones. MDs were originally proposed by Fan (2008) as multirelation constraints, with Schirmer et al. (2020) presenting a MDs discovery system that restricts them to a single relation. When restricted to a single relation, any MD can be represented by a similarity DC.

Another example of IC that handles similarity is inclusion dependencies (sINDs), which are proposed by Kaminsky et al. (2023) and relax inclusion dependencies (INDs) using similarity. sINDs represent constraints of a different nature from DCs. DCs enforce consistency between pairs of tuples, while sINDs enforce the inclusion of values from one attribute in another.

Differential dependencies (DDs) (Song and Chen, 2011; Kuang et al., 2024) also relax restrictions to deal with noisy data. They do it by including the quantification of differences in its conditions definition, allowing textual values to be considered similar until a difference threshold. DDs and DCs with similarity operators share much of the domain of ICs they can represent, but DCs allow constraints with conditions comparing different attributes, which is not supported by DDs.

In DCs, most of the works are restricted to the traditional set of exact operators (Chu et al., 2013; Pena and de Almeida, 2018; Pena et al., 2019, 2020, 2022; Liu et al., 2024). Some works use relaxed definitions, theoretically allowing for similarity DCs, but none of them further explore this subject (Bertossi, 2011; Rekatsinas et al., 2017; Giannakopoulou et al., 2020). HoloDetect (Heidari et al., 2019) uses embeddings to detect violations of DCs with similarity operators, but it requires training with labeled data, which is not necessary for SimFACET.

As presented, neither of these ICs can be used to detect all violations detected when using similarity DCs. Including similarity operators on DCs allows for the detection of violations

that can not be detected using any of the discussed IC types. Therefore, SimFACET advances the capacity of detecting constraint violations in noisy data and represents an advance in the data cleaning area. Table 6.1 presents a summary of the comparison between multiple IC types and similarity DCs.

Table 6.1 - Different ICs compared with similarity DCs

IC type	Supports Similarity?	Can be fully expressed by traditional DCs?	Can be fully expressed by similarity DCs?	Can fully express similarity DCs?
UCCs	✗	✓	✓	✗
FDs	✗	✓	✓	✗
CFDs	✗	✓	✓	✗
ODs	✗	✓	✓	✗
INDs	✗	✗	✗	✗
MDs	✓	✗	✓	✗
sINDs	✓	✗	✗	✗
DDs	✓	✗	✗	✗

Source: prepared by the author (2026)

6.3 SIMILARITY JOIN

As the processing of the similarity refinements requires a similarity join, we investigate both the literature on similarity indexes and on similarity joins. We divide the works according to the similarity metric they use. First, we present indexes and algorithms for edit distance between strings (Bocek et al., 2007; Li et al., 2011; Zhang and Zhang, 2017, 2019; Karpov et al., 2023). After, we present indexes and algorithms for cosine distance between text embeddings (Weber et al., 1998; Böhm et al., 2001; Jégou et al., 2011; Anastasiu and Karypis, 2014; Malkov and Yashunin, 2020; Santana and Ribeiro, 2023; Facebook, 2025).

6.3.1 Edit Distance

According to the results presented by Jiang et al. (2014), FastSS (Bocek et al., 2007) and PassJoin (Li et al., 2011) are the fastest algorithms for string similarity joins using edit distance, each of them being more suited for string collections with different characteristics. FastSS is a string similarity search algorithm for large string collections. String similarity search is the operation of, given one string, finding all strings similar to it in a string collection, so its extension to a string similarity join algorithm is very direct. FastSS is faster than other algorithms evaluated when executed over collections with small average string size, but leads to an accentuated increase in execution time as the average size grows. PassJoin, on the other hand, scales well with the average string size, but is not as fast as FastSS when dealing with collections where the average string size is small.

After the survey presented by Jiang et al. (2014), some other algorithms have been proposed to perform similarity search and join using edit distance, but none of them have the same characteristics of FastSS and PassJoin. MinJoin (Zhang and Zhang, 2019) and MinJoin++ (Karpov et al., 2023) do not guarantee the return of all similar string pairs. EmbedJoin (Zhang and Zhang, 2017) is designed to deal with very large string collections,

where the average string size is larger than 400 characters, which is not the category of textual data we are interested in this work.

In SimFACET, we decided to use FastSS for two main reasons. First, the string length range where FastSS is efficient is similar to the range of categorical columns, an important use case for similarity in constraints. Second, we decide to use an exact approach, as it is already fast enough for violations detection, including being faster than our approaches for cosine distance similarity join.

6.3.2 Cosine Distance

A common indexing technique for improving the similarity search on vectors is the use of trees (Weber et al., 1998; Böhm et al., 2001; Jégou et al., 2011), including R-trees, R*-trees, X-trees, TV-trees, and KD-trees. However, most of the embedding models generate vectors with high dimensionality, in a way that the *curse of dimensionality* makes these tree-based techniques inefficient for text embeddings (Weber et al., 1998; Böhm et al., 2001; Jégou et al., 2011).

Some alternatives for high-dimensional vectors have been proposed. Bayardo et al. (2007) presents the All-Pairs algorithm, which uses pruning strategies and dynamic indexing with prefix filtering to reduce the execution time of embeddings similarity join. Many other algorithms follow and improve the ideas of All-Pairs. One of these algorithms is L2AP (Anastasiu and Karypis, 2014), which uses a new prefix filtering strategy based on ℓ^2-norm to reduce the index size and speed up the join. However, both All-Pairs and L2AP are designed for sparse vectors, while text embeddings are commonly dense vectors, so that their optimizations have lower or no impact on the execution time of string similarity join of text embeddings (Santana and Ribeiro, 2023).

An indexing technique more tolerant to the high dimensionality and density of text embeddings is locality sensitive hashing (LSH) indexes (Indyk and Motwani, 1998; Facebook, 2025). This hash-based approximate indexing approach consists of using multiple hashing functions to group vectors in buckets in a way that similar vectors are mapped to the same bucket. Experimental evaluation, however, shows that LSH indexes suffer from relevant accuracy issues (Jégou et al., 2011).

Another approximate indexing technique is inverted file (IVF) indexes (Sivic and Zisserman, 2003; Facebook, 2025). It consists of clustering vectors using k-means in a way that, if a query vector is similar to a centroid, then it is similar to the vectors in the cluster indexed by that centroid. IVF indexes achieve good accuracy and balance fast search with acceptable building time (Yang et al., 2020).

Finally, we have hierarchical navigable small world (HNSW) indexes (Facebook, 2025; Malkov and Yashunin, 2020; Santana and Ribeiro, 2023). They are approximate indexes based on multilayer graphs that have elevated accuracy and are one of the current faster alternatives in terms of similarity search and join. However, the cost of the low search time is a very high building time (Yang et al., 2020).

SimFACET processes text embedding vectors that are both high-dimensional and dense, so we cannot use tree-based indexes nor All-Pairs, L2AP, or similar algorithms. Besides, as the detection accuracy of similarity DC violations is directly related to the accuracy of the similarity join, the accuracy issues of LSH indexes make them inappropriate for our use case. Both IVF and HNSW indexes have good accuracy and acceptable search time, being used in modern SGBDs as PostgreSQL (Yang et al., 2020) and Milvus (Wang et al., 2021). However, in SimFACET, the index building time is part of the detection time, so the elevated building time

of HNSW indexes makes them prohibitive for us. Therefore, SimFACET uses IVF indexes for performing similarity joins using cosine distance.

6.4 CONCLUSION

In this chapter, we presented and discussed the works related to our research. We started by providing an overview of the data cleaning area, discussing the three main steps in that process, focusing on the detection of violations, the main subject of this work. Then, we compared multiple IC types with similarity DCs, showing that they represent an improvement in the capacity of expressing data rules. Finally, we discussed the existing strategies for performing the similarity joins required for processing similarity predicates.

7 CONCLUSION

In this work, we explored the inclusion of similarity operators in DCs for the detection of violations in noisy data. This inclusion allows for the detection of violations not captured by any other IC type, including those where the inclusion of similarity operators has already been explored. For the detection, we presented `SimFACET`, a system that detects violations of similarity DCs across large datasets in a feasible time. We implemented the support for edit distance between strings and cosine distance between text embeddings as similarity metrics in `SimFACET`. To process similarity operators, we proposed algorithms relying on indexing techniques, exploring approximate alternatives for cosine-based similarity. We also explored the use of different strategies to order the processing of the similarity predicates in the refinements pipeline of `SimFACET`. With our experimental evaluation, we demonstrated that our approximate algorithms reduce detection time compared with exact alternatives while maintaining high accuracy, and extracted insights indicating that processing similarity operators before less-selective operators can reduce detection time in most cases.

From `SimFACET` and our experimental evaluation, we devise some paths for future research about the inclusion of similarity operators in DCs. One of them is to investigate optimizations on the parametrization and sampling in the `CDSampledIVF` algorithm for further reducing the detection time. Another one is developing a heuristic to dynamically decide which strategy adopt for building the refinements pipeline, deciding in execution time the placement of the similarity operators in the pipeline. Similarly, we can dynamically decide which similarity algorithm use based on data characteristics. Another interesting subject is the formal definition and implementation of a non-similarity operator. If defined as the negation of the similarity operator, we can reuse the similarity join algorithms for performing the evaluation of the non-similarity operator. Finally, another interesting topic is the investigation of the impact on detection accuracy of using similarity DCs instead of traditional DCs.

REFERENCES

Abedjan, Z., Chu, X., Deng, D., Fernandez, R. C., Ilyas, I. F., Ouzzani, M., Papotti, P., Stonebraker, M., and Tang, N. (2016). Detecting data errors: where are we and what needs to be done? *Proc. VLDB Endow.*, 9(12):993–1004.

Anastasiu, D. C. and Karypis, G. (2014). L2ap: Fast cosine similarity search with prefix l-2 norm bounds. In *2014 IEEE 30th International Conference on Data Engineering*, pages 784–795.

Bayardo, R. J., Ma, Y., and Srikant, R. (2007). Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 131–140, New York, NY, USA. Association for Computing Machinery.

Bertossi, L. (2011). *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers.

Bertossi, L. and Chomicki, J. (2004). *Query Answering in Inconsistent Databases*, pages 43–83. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bian, L., Yang, W., Xu, J., and Tan, Z. (2024). Discovering denial constraints based on deep reinforcement learning. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, CIKM '24, page 120–129, New York, NY, USA. Association for Computing Machinery.

Bleifuß, T., Kruse, S., and Naumann, F. (2017). Efficient denial constraint discovery with hydra. *Proc. VLDB Endow.*, 11(3):311–323.

Bocek, T., Hunt, E., and Stiller, B. (2007). *Fast similarity search in large dictionaries*. University of Zurich.

Böhm, C., Berchtold, S., and Keim, D. A. (2001). Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373.

ChicagoDataPortal (2025). Food inspections.

Chu, X., Ilyas, I. F., and Papotti, P. (2013). Discovering denial constraints. *Proc. VLDB Endow.*, 6(13):1498–1509.

Consonni, C., Sottovia, P., Montresor, A., and Velegrakis, Y. (2019). Discovering order dependencies through order compatibility. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 409 – 420.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. In Burstein, J., Doran, C., and Solorio, T., editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Facebook (2025). Faiss.

Fan, W. (2008). Dependencies revisited for improving data quality. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, page 159–170, New York, NY, USA. Association for Computing Machinery.

Fan, W., Geerts, F., Jia, X., and Kementsietsidis, A. (2008). Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2).

Flajolet, P., Fusy, É., Gandouet, O., and Meunier, F. (2007). HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In Jacquet, P., editor, *DMTCS Proceedings*, volume DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07) of *DMTCS Proceedings*, pages 137–156, Juan les Pins, France. Discrete Mathematics and Theoretical Computer Science.

Geerts, F., Mecca, G., Papotti, P., and Santoro, D. (2013). The llunatic data-cleaning framework. *Proc. VLDB Endow.*, 6(9):625–636.

Giannakopoulou, S., Karpathiotakis, M., and Ailamaki, A. (2020). Cleaning denial constraint violations through relaxation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 805–815, New York, NY, USA. Association for Computing Machinery.

Heidari, A., McGrath, J., Ilyas, I. F., and Rekatsinas, T. (2019). Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 829–846, New York, NY, USA. Association for Computing Machinery.

Heise, A., Quiané-Ruiz, J.-A., Abedjan, Z., Jentzsch, A., and Naumann, F. (2013). Scalable discovery of unique column combinations. *Proc. VLDB Endow.*, 7(4):301–312.

HuggingFace (2025). paraphrase-mpnet-base-v2.

Huhtala, Y., Kärkkäinen, J., Porkka, P., and Toivonen, H. (1999). Tane: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111.

Indyk, P. and Motwani, R. (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 604–613, New York, NY, USA. Association for Computing Machinery.

Jiang, Y., Li, G., Feng, J., and Li, W.-S. (2014). String similarity joins: an experimental evaluation. *Proc. VLDB Endow.*, 7(8):625–636.

Jégou, H., Douze, M., and Schmid, C. (2011). Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128.

Kaminsky, Y., Pena, E. H. M., and Naumann, F. (2023). Discovering similarity inclusion dependencies. *Proc. ACM Manag. Data*, 1(1).

Karpov, N., Zhang, H., and Zhang, Q. (2023). Minjoin++: a fast algorithm for string similarity joins under edit distance. *The VLDB Journal*, 33(2):281–299.

Khayyat, Z., Lucia, W., Singh, M., Ouzzani, M., Papotti, P., Quiané-Ruiz, J.-A., Tang, N., and Kalnis, P. (2015). Lightning fast and space efficient inequality joins. *Proc. VLDB Endow.*, 8(13):2074–2085.

Kossmann, J., Papenbrock, T., and Naumann, F. (2021). Data dependencies for query optimization: a survey. *The VLDB Journal*, 31(1):1–22.

Kuang, S., Yang, H., Tan, Z., and Ma, S. (2024). Efficient differential dependency discovery. *Proc. VLDB Endow.*, 17(7):1552–1564.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707–710.

Li, G., Deng, D., Wang, J., and Feng, J. (2011). Pass-join: a partition-based method for similarity joins. *Proc. VLDB Endow.*, 5(3):253–264.

Liu, Z., Deep, S., Fariha, A., Psallidas, F., Tiwari, A., and Floratou, A. (2024). Rapidash: Efficient detection of constraint violations. *Proc. VLDB Endow.*, 17(8):2009–2021.

Livshits, E., Kochirgan, R., Tsur, S., Ilyas, I. F., Kimelfeld, B., and Roy, S. (2021). Properties of inconsistency measures for databases. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1182–1194, New York, NY, USA. Association for Computing Machinery.

Luciano, L., Moro, W., de Almeida, E. C., and Pena, E. (2023). Projeção eficiente de violações de restrições de negação. In *Anais do XXXVIII Simpósio Brasileiro de Bancos de Dados*, pages 390–395, Porto Alegre, RS, Brasil. SBC.

Mahdavi, M. and Abedjan, Z. (2020). Baran: effective error correction via a unified context representation and transfer learning. *Proc. VLDB Endow.*, 13(12):1948–1961.

Malkov, Y. A. and Yashunin, D. A. (2020). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. In Bengio, Y. and LeCun, Y., editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*.

NYCOpenData (2025a). Dob job application filings.

NYCOpenData (2025b). Registration contacts.

Pena, E. H. M. and de Almeida, E. C. (2018). Bfastdc: A bitwise algorithm for mining denial constraints. In *Database and Expert Systems Applications: 29th International Conference, DEXA 2018, Regensburg, Germany, September 3–6, 2018, Proceedings, Part I*, page 53–68, Berlin, Heidelberg. Springer-Verlag.

Pena, E. H. M., de Almeida, E. C., and Naumann, F. (2019). Discovery of approximate (and exact) denial constraints. *Proc. VLDB Endow.*, 13(3):266–278.

Pena, E. H. M., de Almeida, E. C., and Naumann, F. (2021). Fast detection of denial constraint violations. *Proc. VLDB Endow.*, 15(4):859–871.

Pena, E. H. M., Lucas Filho, E. R., de Almeida, E. C., and Naumann, F. (2020). Efficient detection of data dependency violations. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, CIKM '20, page 1235–1244, New York, NY, USA. Association for Computing Machinery.

Pena, E. H. M., Porto, F., and Naumann, F. (2022). Fast algorithms for denial constraint discovery. *Proc. VLDB Endow.*, 16(4):684–696.

Reimers, N. and Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In Inui, K., Jiang, J., Ng, V., and Wan, X., editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China. Association for Computational Linguistics.

Rekatsinas, T., Chu, X., Ilyas, I. F., and Ré, C. (2017). Holoclean: holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201.

Santana, D. and Ribeiro, L. (2023). Approximate similarity joins over dense vector embeddings. In *Anais do XXXVIII Simpósio Brasileiro de Bancos de Dados*, pages 51–62, Porto Alegre, RS, Brasil. SBC.

Schirmer, P., Papenbrock, T., Koumarelas, I., and Naumann, F. (2020). Efficient discovery of matching dependencies. *ACM Trans. Database Syst.*, 45(3).

Sivic and Zisserman (2003). Video google: a text retrieval approach to object matching in videos. In *Proceedings Ninth IEEE International Conference on Computer Vision*, pages 1470–1477 vol.2.

Smile (2025). Clustering.

Song, S. and Chen, L. (2011). Differential dependencies: Reasoning and discovery. *ACM Trans. Database Syst.*, 36(3).

Tamalu, N., Ensina, L., de Almeida, E. C., Pena, E., and Oliveira, L. (2023). Fault detection in transmission lines: a denial constraint approach. In *Anais do XXXVIII Simpósio Brasileiro de Bancos de Dados*, pages 231–243, Porto Alegre, RS, Brasil. SBC.

Wang, J., Yi, X., Guo, R., Jin, H., Xu, P., Li, S., Wang, X., Guo, X., Li, C., Xu, X., Yu, K., Yuan, Y., Zou, Y., Long, J., Cai, Y., Li, Z., Zhang, Z., Mo, Y., Gu, J., Jiang, R., Wei, Y., and Xie, C. (2021). Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2614–2627, New York, NY, USA. Association for Computing Machinery.

Weber, R., Schek, H.-J., and Blott, S. (1998). A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, page 194–205, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Wyss, C., Giannella, C., and Robertson, E. L. (2001). Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract. In *Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery*, DaWaK '01, page 101–110, Berlin, Heidelberg. Springer-Verlag.

Yang, W., Li, T., Fang, G., and Wei, H. (2020). Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2241–2253, New York, NY, USA. Association for Computing Machinery.

Zhang, H. and Zhang, Q. (2017). Embedjoin: Efficient edit similarity joins via embeddings. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 585–594, New York, NY, USA. Association for Computing Machinery.

Zhang, H. and Zhang, Q. (2019). Minjoin: Efficient edit similarity joins via local hash minima. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 1093–1103, New York, NY, USA. Association for Computing Machinery.