Inconsistency-aware quantification, query answering and ranking in relational databases

Ousmane Issa

Université Clermont Auvergne

École Doctorale de Sciences Pour L'Ingénieur

Doctoral Thesis

Inconsistency-aware Quantification, Query Answering and Ranking in Relational Databases

A thesis submitted in fulfillment of the requirements for the degree of Doctor of University Clermont Auvergne

Author:
Ousmane ISSA

Supervisors:
Farouk TOUMANI
and
Angela BONIFATI

Jury:

M. François Goasdoué,	Professor,	*Reviewer*
Mme. Laure Berti,	Research Director,	*Reviewer*
Mme. Stefania Dumbrava,	Associate Professor,	*Examiner*
M. Farouk Toumani,	Professor,	*Supervisor*
M. Angela Bonifati,	Professor,	*Supervisor*

LIMOS, UMR-6158, CNRS
Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systmes
Clermont-Ferrand, France

21/02/2022

Abstract

The inconsistency problems in databases and knowledge bases have been largely tackled and discussed in the last forty years. Inconsistency is one of the main dimensions of data quality. In our era, data is the new gold, but data without quality or lack of quality measures is another burden leading to erroneous and uninformative analysis results from data. The inconsistency problem arises when a set of constraints that have to be satisfied by the database instance are violated by this database instance. All the previous works that deal with the problem of inconsistency are focused on either the repair of the inconsistent database to obtain a new database that is consistent (i.e, there is no violation of constraints), or quantification of the inconsistency in the entire database. In this thesis, we propose a new approach to handle inconsistency in relational database by quantifying it on the level of tuples, and then ranking tuples/answers according to their inconsistency to enable choosing among query answers the most consistent/inconsistent ones. So, we define different new of measures of inconsistency degrees that based either on tuples violation (tuple-based approach) or on constraints violation(constraint-based approach). We consider the class of denial constraints as class of constraints and the class of conjunctive queries as class of queries. We leverage why-provenance and polynomial provenance to identify inconsistent tuples and to compute inconsistency degrees of query answers, respectively. We convert each denial constraint into a boolean conjunctive query and evaluate this last one on database to compute the why-provenance of the *true* answer. Using why-provenance, each tuple in the database is annotated with the set of constraints that it violates and its identifiers in a monomial form (otherwise, i.e, the tuple does not involve in violation of any constraint, then it is annotated by the monomial 1), then we obtain an annotated database. Given a conjunctive query Q, Q is evaluated on the annotated database and each answer is computed with a polynomial provenance that encodes in a polynomial formula the set of constraints violated by the answers as well as the set of tuples used to compute answer and involved in violation of these constraints. Then, we define twelve measures of inconsistency degrees using the polynomial provenance of answers. Once, measures of inconsistency are defined, it is interesting to allow ranking of answers (tuples in database) according to their inconsistency degrees. We design a set of top-k algorithms, including TopINC on which the idea of other algorithms is based, allowing to rank the query answers according to their inconsistency degrees. We introduce a new class of algorithms with a new cost model and shown the optimality of these top-k algorithms in some specifics conditions. Also, for each top-k algorithm, we give its theoretical complexity. We have conducted a large experiment to show the feasibility of our approach in practice and also to show the efficiency of our top-k developed algorithms.

Key words: *Inconsistency, Inconsistency Measure, Top-K Algorithm, Provenance, Conjunctive Query, Denial Constraint*

Résumé

Les problèmes de l'incohérence dans les bases de données et les bases de connaissances ont été largement abordés et discutés au cours des quarante dernières années. L'incohérence est l'une des principales dimensions de la qualité des données. À notre époque, les données sont le nouvel or, mais les données sans qualité ou l'absence de mesures de qualité peuvent entraîner d'autres fardeaux qui conduisent à des résultats d'analyse erronés et peu informatifs à partir des données. Le problème de l'incohérence survient lorsqu'un ensemble de contraintes qui doivent être satisfaites par l'instance de la base de données sont violées par cette instance. Les travaux précédents qui traitent du problème de l'incohérence se sont intéressés soit de la réparation de la base de données incohérente pour obtenir une nouvelle base de données qui est cohérente (c'est-à-dire qu'il n'y a pas de violation des contraintes), soit sur la quantification de l'incohérence dans la base de données entière. Dans cette thèse, nous proposons une nouvelle approche pour gérer l'incohérence dans les bases de données relationnelles en la quantifiant au niveau des tuples, puis en classant les tuples/réponses selon leur incohérence pour permettre de choisir parmi les réponses aux requêtes celles qui sont les plus cohérentes/inconsistantes. Ainsi, nous définissons différentes nouvelles mesures de degrés de l'incohérence basées soit sur la violation des tuples. Nous considérons la classe des contraintes de déni(denial constraint en anglais) et la classe des requêtes conjonctives. Nous tirons parti des méthodes why-provenance et polynomial provenance pour identifier les tuples incohérents et pour calculer les degrés de l'incohérence des réponses aux requêtes, respectivement. Nous convertissons chaque contrainte de déni en une requête booléenne conjonctive et évaluons cette dernière sur la base de données pour calculer le why-provenance de la réponse *true*. En utilisant le why-provenance, chaque ligne de la base de données est annotée avec l'ensemble des contraintes qu'elle viole et son identifiant sous une forme de monôme (dans le cas contraire, c'est-à-dire si la ligne de donées n'est impliquée dans aucune violation de contrainte, elle est alors annotée par le monôme 1), on obtient alors une base de données annotée. Étant donné une requête conjonctive Q, Q est évaluée sur la base de données annotée et chaque réponse est calculée avec une provenance polynomiale qui encode dans une formule polynomiale l'ensemble des contraintes violées par les réponses ainsi que l'ensemble des lignes de données utilisées pour calculer la réponse et impliquées dans la violation de ces contraintes. Ensuite, nous définissons douze mesures de degré de l'incohérence en utilisant la provenance polynomiale des réponses. Une fois les mesures d'incohérence définies, il est intéressant de permettre le classement des réponses aux rêquetes en fonction de leur degré d'incohérence. Nous concevons un ensemble d'algorithmes de top-k, dont TopINC sur lequel est basée l'idée des autres algorithmes, permettant de classer les réponses aux requêtes en fonction de leurs degrés d'incohérence. Nous introduisons une nouvelle classe d'algorithmes avec un nouveau modèle de coût et montrons l'optimalité de ces algorithmes de top-k dans certaines conditions spécifiques. De

plus, pour chaque algorithme de top-k, nous donnons sa complexité théorique. Nous avons mené une grande expérience pour montrer la faisabilité de notre approche en pratique et aussi pour montrer l'efficacité de nos algorithmes de top-k développés.

À la mémoire de mes Papas :
À mon défunt Papa ***Issa OUSMANE****,*
À mon défunt Oncle ***Issiaka NOUHOU****,*
À mon défunt Tonton ***Ousmane DOUMBIA***

«Il n'y a le défaut que s'il est entraitenu».
«Maitriser l'incohérence c'est se rapporcher de la perfection».

«Mais l'incohérence n'est pas le monopole des fous: toutes les idées essentielles d'un homme sain sont des constructions irrationnelles». **André Maurois**

Acknowledgements

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

INTRODUCTION

In the last twenty years, special attention has been given to the problem of data quality [15, 54, 55, 99]. Poor data quality can cause major problems in decision making in private companies and public organizations[57]. Hence, It is essential to alleviate this problem and bring concrete solutions. Nowadays, with the evolution of artificial intelligence, the quality of the data plays an important role in the quality of models and algorithms.

There are several dimensions of data quality[127, 133]. Among which the most important ones are [133]:

- data completeness: it treats the missing data problem; it gives also information about how the real world is entirely represented by data [14].
- data accuracy: it is concerned with the design of a set of tools to ensure the correctness of stored data [14, 145].
- data currency: it is about the data freshness [14, 127].
- data consistency: it concerns the reliability of data in compliance with a set of constraints that have to be satisfied by data [14]. Integrity constraints are notable class of such constraints.

This thesis focuses on data inconsistency. There is inconsistency in data when at least one of the set of these constraints, that have to be satisfied, is violated by these data. Inconsistency can be introduced in database for various reasons, among these reasons we have:

- data integration, that consists to combine (heterogeneous) data from many sources [46, 100],
- temporary disabling of constraints checking in relational databases [46],
- application of new constraints to stored data in databases [46],
- human mis-typing during data entry in database,
- inconsistency in set of constraints defined on databases, etc..

Several research problems around inconsistency have been conducted such as fixing the inconsistency problem in knowledge databases[17, 18, 38, 79], database repairing and consistent query answers[9, 22], data cleaning [36], constraints checking (by quantifying inconsistency, for example, before any update operation in databases) [46].

However, little attention has been paid to leaving the database instances intact and quantifying their degrees of inconsistency at different levels of granularity (tuple, sets of tuples, attribute, set of attributes, etc..). Such a characterization enables the users of a DBMS to quantify the level of trust that they shall expect from the data that they query and manipulate. In our work, we are interested in augmenting relational instances with novel inconsistency measures that can also be propagated to query results. The inconsistency degrees of an answer of a given query is determined by relying on provenance-based information of the input tuples involved in the computation of this answer. We first leverage why-provenance in order to identify the inconsistent base tuples of a relational instance with respect to a set of denial constraints. Then, we rely on provenance polynomials [74] in order to propagate the annotations of inconsistencies from the base tuples to the answer tuples of conjunctive queries. Building upon the computed annotations, we define twelve measures of inconsistency degrees, which consider single and multiple violations of constraints and tuples. Since some of our measures are non-monotonic functions, we design new top-k algorithms to rank the top-k results of a query w.r.t. the inconsistency measures, as presented in table 1.1. Since the existing cost models are not suited to our context, we introduce a new class of algorithms called SBA and a new cost function denoted $cost^{\Delta}$ tailored to generic scoring function(monotonic and non monotonic). We show the optimality our top-k algorithms in SBA w.r.t the cost function $cost^{\Delta}$.

We envision several applications of our framework, as follows.
Inconsistency-aware queries for analytical tasks, as we expect that our framework enables inconsistency quantification in querying and analytical tasks within data science pipelines. Our annotations are not merely numbers and convey provenance-based information about the violated constraints, the latter being viable for user consumption in data science tasks.
External annotations for data cleaning pipelines, as our approach can also ease data cleaning tasks in tools such as OpenRefine, Wrangler and Tableau by injecting into them the external information of inconsistency indicators and putting upfront the resulting ranking prior to cleaning and curation.
Approximation schemes for integrity constraints that have been used in order to guarantee a polynomial number of samples in recent probabilistic inference approaches for data cleaning [128]. We believe that an alternative to constraint approximation would be to build samples based on the top-k number of constraints leading to the most consistent (the least consistent, respectively) tuples.
Combined ranking as our quality-informed ranking can be combined with other ranking criteria, e.g. user preferences in recommender systems and unfairness and discrimination in marketplaces and search sites [5].

Contributions

We introduce a novel framework to manage inconsistency in relational database. As shown in figure 1.1, we consider as an input of the framework a database and a set of denial constraints, then, we deploy two-steps process:

- **Preprocessing step,** we convert this input database, according to the set of denial constraints, into another database where each tuple is annotated with the set of constraints it violates. We rely on why-provenance [49, 73] to compute such annotations.

Constraint-based measures		
Measure	**Monotonic**	**Top-k algorithms**
CBM	✓	NA, TopIncMem, TopINC
CBS	✗	NA, TopINC
CSM_{min}	✓	NA, TopMultiSet, TopINCDE
CSM_{max}	✓	
CSS_{min}	✗	NA, TopIncSet, TopINCDE
CSS_{max}	✗	
Tuple-based measures		
TBM	✓	NA, TupIncRank
TBS	✗	NA
TSM_{min}	✓	NA, TopMultiSet, TopINCDE
TSM_{max}	✓	
TSS_{min}	✗	NA
TSS_{max}	✗	

Table 1.1: Measures with their associated top-k algorithms

- **Query evaluation step,** we consider as class of queries the class of conjunctive queries. The queries are evaluated on the converted database using polynomial provenance [74]. In the step of query evaluation, query answers can be ranked according to their inconsistency degrees and the k first interesting answers (the most consistent/inconsistent) are chosen using one of the top-k algorithms defined in this thesis.

In this thesis we make the following technical contributions:

1. We design novel measures of inconsistency degrees of answer for conjunctive queries over an inconsistent database in the presence of a set of denial constraints 4.4. In particular, we consider two approaches to quantify inconsistency of query answers:
   - **Tuple-based approach,** let t be a query answer, this class of measures counts the number of inconsistent tuples involved in computation of t. We propose six different measures of inconsistency of query answers in this class [86].
   - **Constraint-based approach,** considering t a query answer, this class of measures counts the number of constraints violated by the tuples involved in computation of t. We proposed six measures of inconsistency of query answers in this class [87].
2. We define top-k problems and threshold problems based on these measures [87].
   - We classify the top-k algorithms into two classes, Disk-based algorithms and Memory-based algorithms, according to the nature of the cost model that they minimize.
   - We design new top-k algorithms to efficiently compute top-k answers in the context where the input scoring functions are the set of measures defined above.
3. We theoretically show the efficiency of these algorithms

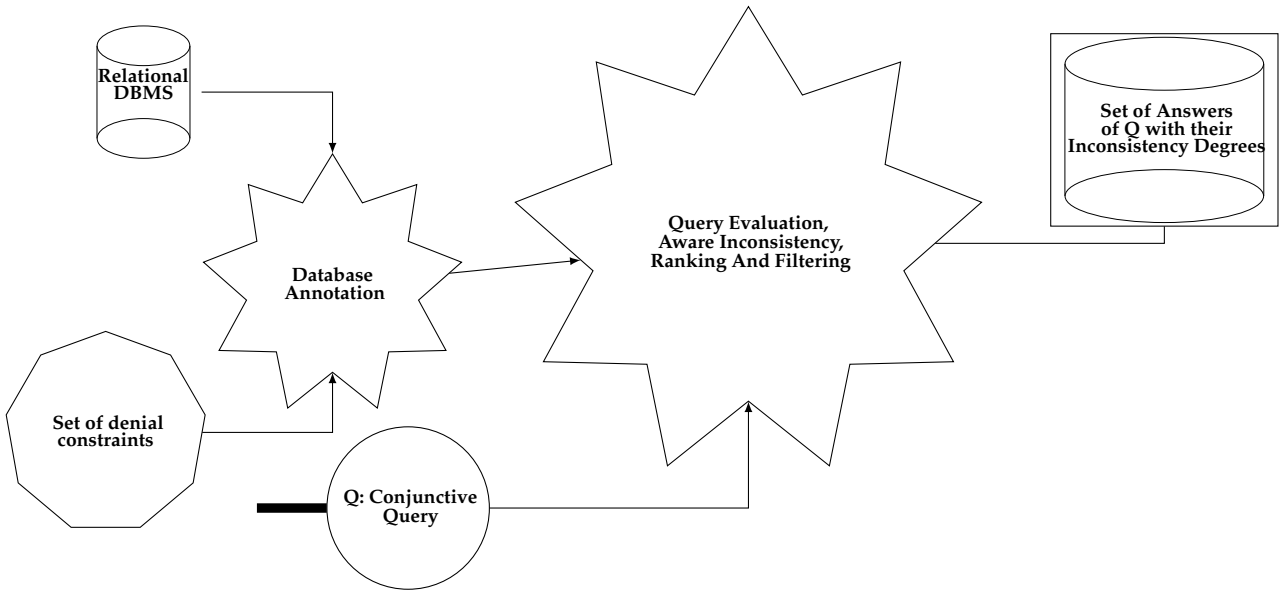


Figure 1.1: Framework aware inconsistency in relational database

- We define a new class of algorithms called semi-blind algorithms (SBA) that contains both algorithms working with monotonic scoring functions and those working with non-monotonic scoring functions.
- We introduce a new cost model denoted $cost^{\Delta}$ that minimizes the number of tuples read on disk while avoiding indeterminism caused by the tuples having the same score but different values in join attributes.
- We show optimality of these top-k algorithms in SBA class w.r.t our new cost model $cost^{\Delta}$ for algorithms in Disk-based. For the algorithms in Memory-based, we show that they are polynomial in data complexity.

4. We experimentally show efficiencies of our main algorithm [87].
   - The running time of some measures is studied and analyzed.
   - The time consumed by the preprocessing phase is also analyzed.
   - We experimentally lead a quality evaluation of some measures.
5. We develop a tool, called INCA system, that allows users to explore data profiling and query answering profiling based on inconsistency quantification.
6. We present a preliminary work to handle inconsistency in a setting where the set of constraints are inconsistent.

Organization

This thesis is organized as follows: Chapter 2 presents the background; Chapter 3 discusses the state of art; Chapter 4 introduces the set of measures of inconsistency degrees proposed in this thesis; Chapter 5 presents different top-k algorithms for the proposed inconsistency measures; Chapter 6 extends measures of inconsistency with new measures in the context of presence of inconsistency in the set of constraints; Chapter 7 is dedicated to the experimentations and evaluations of the proposed measures and top-k

algorithms; Chapter 8 explores our inconsistency prototype INCA system; Chapter 9 concludes this thesis and gives some perspectives.

In the next chapter we describe the necessary background to this thesis in database management field.

Chapter 2

PRELIMINARY NOTIONS

This chapter introduces some basic notions and notations used throughout this thesis. First, we recall some relevant concepts in relational database and then we present different notions in top-k processing and optimal join processing.

2.1 Relational Database

We assume that the reader is familiarized with relational database [4].

We provide the formal definition of a relational database [4] and different notations used throughout this thesis. The domain of the database is an infinite set of constants, denoted by $\mathcal{D}$. The database schema, denoted $\mathcal{S} = \{R_1, \ldots, R_n\}$ is a finite set of relations/predicates such that $\mathcal{D} \cap \mathcal{S} = \emptyset$. Let **SetAttrs** be a countable infinite set of attribute names such that $\mathcal{S} \cap \textbf{SetAttrs} = \emptyset$ and $\mathcal{D} \cap \textbf{SetAttrs} = \emptyset$. On each relation $R \in \mathcal{S}$ the function $Attr$ associates a set of attributes from **SetAttrs** (i.e, $Attr : \mathcal{S} \rightarrow \mathcal{P}^{fin}(\textbf{SetAttrs})$, with $\mathcal{P}^{fin}(\textbf{SetAttrs})$ the finitary powerset of **SetAttrs**). Given a relation R, then $Attr(R)$ is called the set of attributes of R. A tuple t is a function from a finite subset $A \in \mathcal{P}^{fin}(\textbf{SetAttrs})$ to the domain $\mathcal{D}$; we say that t is a tuple from set of attributes A. A relation $R \in \mathcal{S}$ is a finite set of tuples from $Attr(R)$. We denote by $R(t)$ with t a tuple to mean that the tuple t is in a relation R, denoted also by $t \in R$. A relational database (or shortly database or instance) from a schema $\mathcal{S}$ and a domain $\mathcal{D}$, is a finite set of relations from the schema $\mathcal{S}$ [4]. A relation in a database instance $\mathcal{I}$ is denoted by $\mathcal{I}(R)$. Let id be a function that associates to each tuple an unique identifier, so the identifier of a tuple t is denoted $id(t)$. We denote Γ by the set of all identifiers of the instance of the database considered.

Example 2.1. *Consider the relational schema education consisting of the following relations education $= \{Student, Course\}$ and the domain $\mathcal{D}$ is from the (infinite) set of strings. We have $Attr(Student) = \{Register\_Number, Name, Pref\_Course\}$ and $Attr(Course) = \{Course\_ID, title\}$. An example of a database instance from education and $\mathcal{D}$ is*

REGISTER_NUMBER	NAME	PREF_COURSE
*M*01	*Alice*	*C*02
*M*02	*Bob*	*C*01

COURSE_ID	TITLE
*C*01	*Computer Science*
*C*02	*Physics*
*C*02	*Biology*

The first relation Student records students with their preferred courses and the second relation (Course) records courses.

2.2 Conjunctive Query

In the following, we introduce conjunctive query, i.e, the main class of queries used in this thesis. The class of conjunctive queries ($\mathcal{CQ}$) is defined as a set of queries of the following form:

$$Q(u) \leftarrow R_1(u_1), ..., R_n(u_m), \phi(u_1, \ldots, u_m) \tag{2.1}$$

where each R_i is a relation symbol in $\mathcal{S}$ and Q is a relation symbol in the output schema $\mathcal{O}$, each u_i is a tuple of variables or/and constants having the same arity as R_i, and u is a tuple of distinguished variables. Those are variables occurring in query in tuples u_i with $i \in \{1, \ldots, m\}$, i.e, $Var(u) \subseteq \bigcup_{i \in \{1,...,m\}} Var(u_i)$ with Var a function that returns the set of variables in a tuple) or/and constants. The formula $\phi(u_1, \ldots, u_m)$ is a conjunction of built-in atom under the form $x\ op\ y$ where $op \in \{=, \neq, \geq, \leq, <, >\}$ (op is a arithmetical predicate), with x and y being either variables from $\bigcup_{i \in \{1,...,m\}} Var(u_i)$ or constant. The set of variable in a query Q is denoted $Vars(Q)$. When the conjunctive query does not contain a built-in part or the built-in part contains only equal (=) predicates then it is called an equi-conjunctive query. Any equi-conjunctive query can be converted into an equi-conjunctive query that contains no built-in formula (each built-in predicate $x = y$ is eliminated by rename each instance of y to x). We say that Q is a full conjunctive query if $Var(u) = \bigcup_{i=1}^{m} Var(u_i)$.

A valuation of Q over a domain $\mathcal{D}$ is a function $v : Vars(Q) \rightarrow \mathcal{D}$, extended to be the identity on constants, i.e, for each $e \in \mathcal{D}$ then $v(e) = e$. For a tuple $t = (a_1, \ldots, a_p)$ consisting of variables and/or constraints, $v(t) = (v(a_1), \ldots, v(a_p))$. The result (or query answers) $Q(\mathcal{I})$ of executing a query Q over an instance $\mathcal{I}$ is: $Q(\mathcal{I}) = \{v(u) | v$ is a valuation over $Vars(Q)$ and $\forall i \in [1, n], v(u_i) \in \mathcal{I}(R_i)$ and $\phi(v(u_1), \ldots, v(u_m))$ is true $\}$.

An union of conjunctive queries, denoted **UCQ**, is a set of conjunctive queries with the same output predicate. Let Q be a conjunctive query, Q is a self-join conjunctive query if there are at least two relation atoms in Q, R_i and R_j, such that $R_i = R_j$, $i \neq j$ and $i, j \in [1, m]$. The query Q is a free self-join conjunctive query if Q is not a self-join conjunctive query.

Example 2.2. *Consider the following query* $Stu\_Cour(name, title)$ *:-*$Student(mat, name, Cour_1D)$, $Course(Cour_1D, title)$, $title \neq$ *"Physics". The query Stu_Cour extract all the student names with their preferred courses, only for courses different from "Physics".*

NAME	TITLE
Bob	*Computer Science*

Only one valuation v ($v(name) =$ *"Bob"*, $v(title) =$ *"Computer Science"*, $v(mat) =$ *"M02"*, $v(Cour_1D) =$ *"C01"*) *satisfies the query Stu_Cour, any other valuation fails to satisfy this query.*

2.3 Denial Constraints

Various classes of database constraints have been studied in the context of relational database [16, 37, 56]. In this thesis, we consider the class of denial constraints that is the one of the most used classes of constraints [126]. The class of denial constraints

covers many other classes of constraints such as functional dependencies, conditional functional dependencies, metric functional dependencies [37, 128]. Informally, a denial constraint expresses a forbidden pattern in database. The class of denial constraints ($\mathcal{DC}$) is defined as the set of constraints of the following form:

$$\leftarrow R_1(u_1) \wedge ... \wedge R_n(u_m) \wedge \phi(u_1, \ldots, u_m) \tag{2.2}$$

where the $R_i(u_i)$ are defined as previously, in section 2.2 and in Equation 2.1, and ϕ is a conjunction of a built-in atoms. Let C be a denial constraint, we denote by $Var(C)$ all the variables present in C. We denote by C_{id} an unique identifier of C. We denote Y the set of the identifiers of the denial constraints considered.

A denial constraint C is satisfied by an instance $\mathcal{I}$, denoted $\mathcal{I} \models C$ (otherwise, we denote $\mathcal{I} \not\models C$), if and only if for any valuation v over $Vars(Q)$ then: there exists at least $i \in [1, n], v(u_i) \notin \mathcal{I}(R_i)$ or $\phi(v(u_1), \ldots, v(u_m))$ leads to false. So, we say that the database instance is consistent with respect to the constraint C (or shortly, we say it is consistent). Otherwise, we say that the database is inconsistent.

Let DC be a set of denial constraints over a schema $\mathcal{S}$, DC is consistent if there exists an instance $\mathcal{I} \neq \emptyset$ on schema $\mathcal{S}$ such that $\mathcal{I} \models DC$. Otherwise, we say that DC is inconsistent.

Given a set of denial constraints DC, an instance $\mathcal{I}$ is consistent if and only if $\mathcal{I}$ is consistent with each constraint $C \in DC$ in the set of constraints DC (denoted $\mathcal{I} \models DC$), otherwise we say that the instance $\mathcal{I}$ is inconsistent with respect to the set of denial constraints DC (denoted $\mathcal{I} \not\models DC$).

Example 2.3. *Let $C_1 \leftarrow Student(mat, name, pref\_course), Student(mat, name_1, pref\_course_1)$, $name_1 \neq name$ and $C_2 \leftarrow Course(id, title), Course(id, title_1), title_1 \neq title$ be two denials constraints. The constraint C_1 is about the uniqueness of the student number according to the name and the second constraint (C_2) is about the uniqueness of the course identifier. As one can easily note, the constraint C_1 is satisfied by the instance in example 2.1 since there are no two students with the same number. By opposite, the constraint C_2 is violated by the instance given in Example 2.1 since there are two courses that have the identifier (the identifier C02).*

In the remaining of this thesis, denial constraint is abbreviated constraint, unless otherwise specified.

2.4 Provenance in relational databases

We recall the provenance semiring framework, introduced in [74] as a unifying framework able to capture a wide range of provenance models at different levels of granularity [49, 74, 74]. This framework is based on a general data model that extends the relational model with the so-called *K-relations* in which tuples are assigned annotations from a given semiring. An extension for Postgresql, called **ProvSQL**, enabling to evaluate queries and return answers of queries with their provenances is developed in [132].

A monomial M over $\mathbb{N}$ and a finite set of variables $\mathcal{X}$ is defined by $M = a \times x_1^{m_1} \times ... \times x_n^{m_n}$ with $a, m_1, ..., m_n \in \mathbb{N}$ and $x_1, ..., x_n \in \mathcal{X}$. A null monomial is a monomial where $a = 0$, otherwise we call the monomial a non null monomial. Let $M = a \times x_1^{m_1} \times ... \times x_n^{m_n}$ be a monomial. We denote by $Var(M) = \{x_1, \ldots, x_n\}$ the set of variables that appear in the monomial M. The weight of a variable $x_i \in Var(M)$ w.r.t a monomial M, denoted by $W(M, x_i)$, is equal to m_i, the exponent of the variable x_i in M. The weight

of a non null monomial M, denoted by $W(M)$, is defined as the sum of the weights of its variables, i.e.: $W(M) = \sum_{x \in Var(M)} W(M, x)$. A polynomial P over $\mathbb{N}$ and a finite set of variables $\mathcal{X}$ is a finite sum of monomials over $\mathcal{X}$. We denote by $M(P)$ the set of monomials of P.

Example 2.4. *Consider the following monomials (arbitrarily chosen)*

- $M_1 = 4X_1^2X_2X_3X_4$, $W(M_1) = 5$, $V(M_1) = 4$
- $M_2 = X_1X_2X_3^4$, $W(M_1) = 6$, $V(M_2) = 3$
- $P = M_1 + M_2$, $M(P) = \{M_1, M_2\}$

K-relations

A commutative semiring is an algebraic structure $(K, \oplus, \otimes, 0, 1)$, where 0 and 1 are two constants in K and K is a set equipped with two binary operations $\oplus$ (sum) and $\otimes$ (product) such that $(K, \oplus, 0)$ and $(K, \otimes, 1)$ are commutative monoids[1] with identities 0 and 1 respectively, $\otimes$ is distributive over $\oplus$ and $0 \otimes a = a \otimes 0 = 0$ holds $\forall a \in K$.

Example 2.5. *The following structures are commutative semirings structures:*

- $(\mathbb{N}, +, *, 0, 1)$ *the Natural semiring*
- $(\mathbb{B}, \vee, \wedge, false, true)$ *the Boolean semiring*

Definition 2.1 (K-relations.)**.** *An n-ary K-relation is a function $R : \mathcal{D}^n \to K$ such that its support, defined by $supp(R) \stackrel{def}{=} \{t : t \in \mathcal{D}^n, R(t) \neq 0\}$, is finite.*

Hence, a *K-relation* is an extension of the classical notion of relation to allow tuples to be annotated by the elements of a any semiring but not only by the elements from the Boolean semiring.

Let R be an n-ary *K-relation* and let $t \in \mathcal{D}^n$, the value $R(t) \in K$ assigned to the tuple t by the K-relation R is called the annotation of t in R. Note that $R(t) = 0$ means that t is "out of" R [74]. A *K-instance* is a mapping from relations symbols in a database schema $\mathcal{S}$ to *K-relations* (i.e, a finite set of *K-relations* over $\mathcal{S}$). If $\mathcal{J}$ is a K-instance over a database schema $\mathcal{S}$ and $R_i \in \mathcal{S}$ is a relation symbol in $\mathcal{S}$, we denote by $\mathcal{J}(R_i)$ the K-relation corresponding to the value of R_i in $\mathcal{J}$ (otherwise we use only R_i is $\mathcal{J}$ is understood).

Conjunctive queries on *K-instances*

Let $Q(u) \leftarrow R_1(u_1), ..., R_n(u_m), \phi(u_1, \ldots, u_m)$ be a conjunctive query and let $\mathcal{J}$ be a *K-instance* over the same schema than Q, with $(K, \oplus, \otimes, 0, 1)$ a semiring. A valuation of Q over a domain $\mathcal{D}$ is a function $v : Vars(Q) \to \mathcal{D}$, extended to be the identity on constants. The result of executing a query Q over a *K-instance* $\mathcal{J}$, using the semiring $(K, \oplus, \otimes, 0, 1)$, is the *K-relation* $Q(\mathcal{J})$ defined as follows:

$$Q(\mathcal{J}) \stackrel{\text{def}}{=} \{ \ (v(u), \Pi_{i=1}^{n} R_i(v(u_i))) \mid v \text{ is a valuation over } Vars(Q)\}$$

[1] i.e., $\oplus$ (resp. $\otimes$) is associative and commutative and 0 (resp. 1) is its neutral element.

The *K-relation* $Q(\mathcal{J})$ associates to each tuple $t = v(u)$, which is in the answer of the query Q over the *K-instance* $\mathcal{J}$, an annotation $\Pi_{i=1}^{n} R_i(v(u_i)))$ obtained from the product, using the $\otimes$ operator, of the annotations $R_i(v(u_i))$ of the base tuples contributing to t. Since there could exist different ways to compute the same answer t, the complete annotation of t is obtained by summing the alternative ways (i.e., the various valuations) to derive a tuple t using the $\oplus$ operator. Consequently, the provenance of an answer t of Q over a *K-instance* $\mathcal{J}$ is given by: $Q(\mathcal{J})(t) = \sum_{v \ s.t \ v(u)=t} \Pi_{i=1}^{n} R_i(v(u_i)))$.

A polynomial provenance is the provenance/annotation obtained when the semiring structure considered to evaluate the query is $(\mathbb{N}[X], +, *, 0, 1)$.

Example 2.6. *Consider the database instance in Example 2.1. We associate randomly annotation from the Natural semiring to each tuple as follows, each annotation represents the number of times tuples is present in the instance (in this case, the instance is a bag):*

- $Student(\langle M01, Alice, C02\rangle) = 1$
- $Student(\langle M02, Bob, C01\rangle) = 2$
- $Course(\langle C01, "Computer\ Science"\rangle) = 3$
- $Course(\langle C02, "Physics"\rangle) = 1$
- $Course(\langle C02, "Biology"\rangle) = 2$

Let's denote the previous $\mathbb{N}$*-instance by* $\mathcal{J}$*. Consider now the query in Example 2.2, then we have the unique answer with its annotation is as follows:* $Stu\_Cour(\langle Bob, "Computer\ Science"\rangle)$ $= Student(\langle M02, Bob, C01\rangle) * Course(\langle C01, "Computer\ Science"\rangle) = 6$*. The annotation* 6 *means that this answers can be obtained six times in the* $\mathbb{N}$*-instance* $\mathcal{J}$ *(represented as bag)*

2.5 Hyper Graph and Join Tree

In this section, we overview a set of approaches introduced to efficiently evaluate the relational join operation. The join operator is the expensive operation in query processing. We consider only the equi-conjunctive queries. An equi-conjunctive query (shortly, in the following we use/say query) has the following form :

$$Q(u) \leftarrow R_1(u_1), ..., R_n(u_m)$$

A hypergraph HG is tuple $HG = (V, HE)$ composed from a finite set of nodes (elements from V) and a set of non empty subsets from V called the set of hyper edges (elements in HE).

Definition 2.2 (Query Hyper graph)**.** *Let* $Q(u) \leftarrow R_1(u_1), ..., R_n(u_m)$ *be an equi-conjunctive query, the query hypergraph of* Q*, denoted* $H(Q)$*, is an hypergraph* H *with the set of nodes* $Var(Q)$ *and the set of hyperedges the set* $\{Var(u_i) : i \in \{1, \ldots, m\}\}$*, defined as follows :*

$$H(Q) = (Var(Q), \{Var(u_i) : i \in \{1, \ldots, m\}\})$$

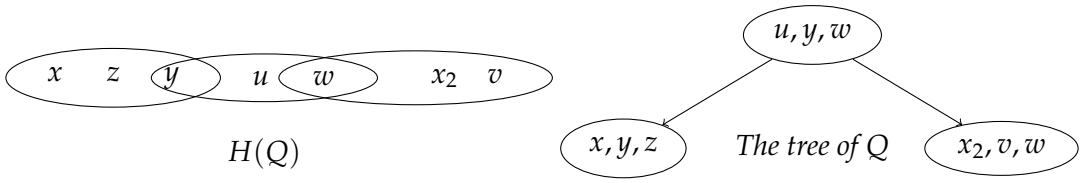
An hypergraph HG is acyclic if and only if it can be turned into a tree T of the following features: the nodes of T are the set of hyperedges and each two n_1 and n_2 in T, with their hyperedges having at least one node from $HG(V)$ in common, so n_1 and n_2 are connected in a sub graph (sub tree) of T [13]. We say that also, an hypergraph

is acyclic if the GYO algorithm (from Graham-Yu-Ozsoyoglu), that takes as input an hypergraph and returns an other hypergraph, outputs an empty hypergraph [158]. The GYO algorithm is as follows:

- Choose on arbitrary node N of the hypergraph $H(Q)$ such that one can divide its set of variables into two parts
  - The first one, denoted $Isolated(N)$, is the set of variables that are present in only N.
  - The other one is the set of variables present in N and only another node.
- Remove N and the set of variables $Isolated(N)$ from the hypergraph $H(Q)$.

A query is acyclic if its underling hypergraph is acyclic.

Example 2.7. *Consider a database $\mathcal{I}$ from this arbitrary schema $\{R_1(A_1, A_2, A_3), R_2(A_4, A_5, A_6), R_3(A_7, A_8, A_9)\}$. Let $Q(x, u, x_2)$:- $R_1(x, y, z)$, $R_2(u, y, w)$, $R_3(x_2, v, w)$ be the query. By GYO algorithm Q is acyclic. The hypergraph $H(Q) = (V, HE)$ where $V = \{x, u, x_2, y, z, v, w\}$ and $HE = \{\{x, y, z\}, \{u, y, w\}, \{x_2, v, w\}\}$.*



The Yannakakis's algorithm [156] designed for acyclic query Q evaluation over an instance $\mathcal{I}$ runs in a complexity $O(|\mathcal{I}| + |Q(\mathcal{I})|)$, it is an optimal. The main lines of this algorithm are:

- Associate to each node of the tree a map. Given a node n, the set of keys of the map of n is from the valuation values of common variables between n and its parent and the its set values is from the tuples of the relation represented by this node;
- The root r node has one key that is the empty tuple since its has no parent, and the set of values corresponding to this key is the set of tuples from relation that r represents;
- Remove the dangling tuples in each node
  - in each parent node remove tuples that do not match any tuple in a least one of its children nodes, using its associated map (top down traversal to remove other dangling tuple);
  - in each node, remove all the tuples that do not match any tuple in its parent node, using the its associated map (bottom up traversal to remove some dangling tuple);
- Answers are computed by doing just a Cartesian product using the result tree

So, any acyclic query can be performed in linear time in data complexity. The following example illustrates the Yanakakis's algorithm.

Example 2.8. *Consider the query and the schema of the example 2.7. Consider the following instance from this schema.*

R_1

$\mathbf{A_1}$	$\mathbf{A_2}$	$\mathbf{A_3}$
a_1	1	3
a_2	2	4

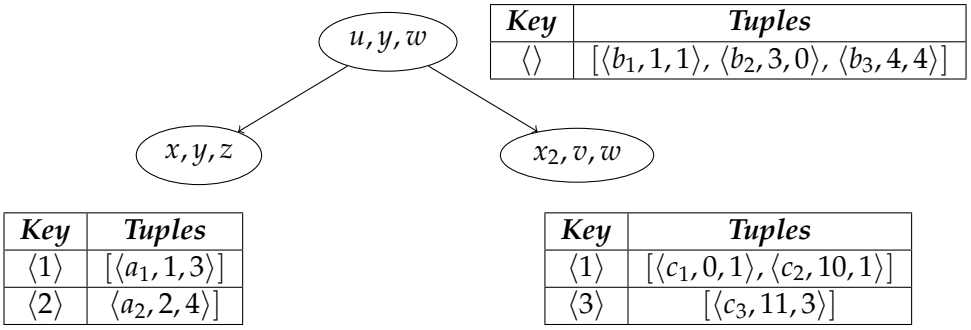
R_2

$\mathbf{A_4}$	$\mathbf{A_5}$	$\mathbf{A_6}$
b_1	1	1
b_2	3	0
b_3	4	4

R_3

$\mathbf{A_7}$	$\mathbf{A_8}$	$\mathbf{A_9}$
c_1	0	1
c_2	10	1
c_3	11	3

The matching between the variables and the attributes is as follows: $x \to \{A_1\}$ $y \to \{A_2, A_5\}$ $z \to \{A_3\}$ $u \to \{A_4\}$ $w \to \{A_6, A_9\}$ $x_2 \to \{A_7\}$ $v \to \{A_8\}$. *On this instance we run the Yannakakis's algorithm as follows:*

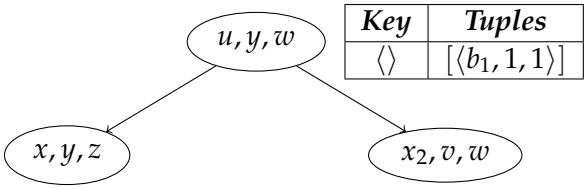


Key	***Tuples***
$\langle\rangle$	$[\langle b_1, 1, 1\rangle, \langle b_2, 3, 0\rangle, \langle b_3, 4, 4\rangle]$

Key	***Tuples***
$\langle 1\rangle$	$[\langle a_1, 1, 3\rangle]$
$\langle 2\rangle$	$\langle a_2, 2, 4\rangle]$

Key	***Tuples***
$\langle 1\rangle$	$[\langle c_1, 0, 1\rangle, \langle c_2, 10, 1\rangle]$
$\langle 3\rangle$	$[\langle c_3, 11, 3\rangle]$

(a.) The tree of Q with data structures



Key	***Tuples***
$\langle\rangle$	$[\langle b_1, 1, 1\rangle]$

Key	***Tuples***
$\langle 1\rangle$	$[\langle a_1, 1, 3\rangle]$

Key	***Tuples***
$\langle 1\rangle$	$[\langle c_1, 0, 1\rangle, \langle c_2, 10, 1\rangle]$

(b.) The tree of Q after dangling tuples eliminations

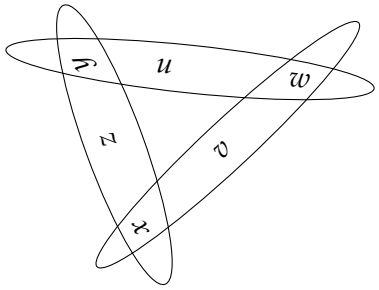
The set of answers is the set $\langle b_1, 1, 1\rangle \times [\langle a_1, 1, 3\rangle] \times [\langle c_1, 0, 1\rangle, \langle c_2, 10, 1\rangle]$ *after projection on variable* x, u, x_2. *So, the set of answers is:*

- $\pi_{x,u,x_2}(\langle b_1, 1, 1\rangle \times \langle a_1, 1, 3\rangle \times \langle c_1, 0, 1\rangle) = \langle a_1, b_1, c_1\rangle$
- $\pi_{x,u,x_2}(\langle b_1, 1, 1\rangle \times \langle a_1, 1, 3\rangle \times \langle c_2, 10, 1\rangle) = \langle a_1, b_1, c_2\rangle$

When the query is not acyclic then, it can be decomposed into a tree in a way that the cycle is removed. This approach merges some nodes of the hypergraph to obtain

an acyclic hypergraph and then obtain the tree [66]. The merged nodes implies intermediary results, i.e, one needs to first compute the result of these merged nodes before computing the final result. Hypergraph decomposition is an active research area. One of the main challenge is to find the hypergraph decomposition that minimizes the intermediary result [66].

Example 2.9. *Consider now the query $Q_1(x,u)$: - $R_1(x,y,z)$, $R_2(u,y,w)$, $R_3(x,v,w)$ similar as the query Q in Example 2.7. As one can easily note, the query Q_1 is a cyclic query since the GYO algorithm does not return an empty result after its running with $H(Q_1) = (V', HE')$ as input hypergraph: any hyperedge among $HE' = \{\{x,y,z\},\{u,y,w\},\{x,v,w\}\}$ has common variables from at least two other hyperedges. The $H(Q_1)$ is as follows:*



$H(Q_1)$

It is possible to merge some hyperedges to have a tree, then an hypergraph that is acyclic. For example, merge the node $\{x,y,z\}$ and $\{u,y,w\}$ to obtain the following $\{\{\{x,y,z,u,w\},\{x,v,w\}\}\}$ hypereadges that forms an acyclic hypergraph. But one has to materialize first the result of join between $\{x,y,z\}$ and $\{u,y,w\}$.

The fractional vertex cover of an hypergraph $H(Q) = (V, HE)$, for a query Q, is a solution of the following system of inequalities [66]:

$$\sum_{e \in HE : v \in e} X_e \geq 1 \ \forall v \in V$$
$$\text{with } X_e \geq 0$$

The variables X_e are real number that represent each hyperedge (an hyperedge corresponds $e \in HE$ to a relational atom in Q). The minimum vertex cover of $H(Q)$ corresponds to the minimum among vertex covers (with summation of their components) of $H(Q)$. So, the minimum vertex cover is obtained by resolving the above system of inequalities with the following objective function to minimize:

$$\sum_{e \in HE} X_e$$

Let $N_R = |R|$ be the number of tuples in the relation R. The AGM bound [12] of the query answers size of a query Q evaluated over an instance $\mathcal{I}$ is $Q(\mathcal{I}) \leq \prod_{i \in \{1,\ldots,m\}} N_{R_i}^{X_{e_i}}$, in the case where $Var(u) = \bigcup_{i \in \{1,\ldots,m\}} Var(u_i)$, with e_i the hyperedge corresponding, in the hypergraph $H(Q)$, to the relational atom R_i. The sum $\sum_{e \in HE} X_e$ is called the fractional width. This maximum fractional width bound allows to choose the best merging of the nodes of a cyclic hypergraph to have a small size of intermediary result. The largest

fractional width among all the merged nodes during the decomposition of the hypergraph to a tree is called the tree fractional width of the resulting tree. The tree fractional width enables to give the largest intermediary result size.

These efficient techniques to compute the join result are used in Chapter 5 to do enumeration of query answers in order of their score (inconsistency degrees according to a measure of inconsistency degrees). The next chapter presents the state of the art.

Chapter 3

STATE OF THE ART

We overview related works in the following areas: inconsistency handling in knowledge bases and relational databases (section 3.1) and top-k queries processing algorithms (3.2).

3.1 Data Inconsistency

We discuss related works in the areas of knowledge bases and relational databases.

3.1.1 Inconsistency in knowledge bases

The problem of inconsistency in knowledge bases is studied since more than forty years ago [38]. In the literature, the inconsistency problem in knowledge bases is handled in two manners: defining new class of logic with new rules of inference that take account inconsistency, called paraconsistent logic system[28, 67, 71, 79, 80, 139, 141] or quantifying the level of inconsistency in the knowledge bases[7, 40, 50, 68, 90, 94, 109, 112–114, 116, 122, 129, 137, 142–144, 153, 159].

The paraconsistent logic systems are defined to tackle the problem of triviality arisen in classical logic caused by the occurrence of inconsistency [10, 17, 18, 24, 25, 154].

To reason in a context of inconsistency in a knowledge bases $\mathbb{K}$, Rescher et al, in [2], propose two notions of logical consequence: one that is strong called W-consequence, in this case a proposition is true if it is a consequence of at least one maximal consistent set (under set-inclusion) of $\mathbb{K}$; the second one called I-consequence, considers all the maximal consistent sets from $\mathbb{K}$ to valid a proposition. Benferhat et al [18] tackle the problem of inconsistency by using argue consequence notion, in other word, any formula valid ϕ using an inconsistent knowledge bases $\mathbb{K}$ is a consequence of a minimal consistent subset (under set-inclusion) $\mathcal{S}$ (i.e, $\mathcal{S} \subseteq \mathbb{K}$ and $\mathcal{S} \models \phi$ and $\nexists \mathcal{S}' \subseteq S$ such that $\mathcal{S}'$ consistent and $\mathcal{S}' \models \phi$) of $\mathbb{K}$; and the contradictory formula of ϕ ($\neg\phi$) is not a consequence of any other minimal consistent subset $\mathcal{S}'$ of $\mathbb{K}$ ($\mathcal{S}' \models \neg\phi$).

Besnard et al [25] introduce a new class of logical system called quasi-classical logic, designed from classical first order logical by making strict inference rules enabling to avoid triviality during deduction (i.e, to avoid to deduce any formula). One of the paraconsistent logic introduced is the four values logic [10, 17]. The four values logic introduces two new truth values. One of the two values concerns formulas that are neither true nor false. The second one is about propositions that are true and false at the same time. An interesting discussion about these paraconsistent systems is done in [28]. These approaches assume a specific logic system to query knowledge bases and

limit query answers to a specific subset of answers, instead of our approaches that keep standard query language and quantify inconsistency of query answers.

Concerning quantification of inconsistency in the knowledge base, a set of measures of inconsistency levels, that measure inconsistency in the entire knowledge bases, are introduced [69, 82, 88, 89, 91, 121, 123, 124, 134, 135, 138]. The inconsistency level of the entire knowledge base enables to compare two knowledge bases [82]. This level of inconsistency can be also used to help the engine of integrity constraint checking [41–47]. Jabbour et al introduce in [91] two measures, measuring inconsistency in whole of the database; one measure counts the number of minimal inconsistent subsets of the knowledge bases and the second counts the number of conflicts(a conflict arises when an atom and its negation are both present in the knowledge base) that one can find in the knowledge bases. Thimm et al propose in [134] a set of measures for probabilistic knowledge bases.

An overview of measures of inconsistency level of knowledge bases in literature is presented in [81]. A study about the quality of the measures of inconsistency level defined in literature is investigated in [70, 136]. All these works quantify inconsistency in the entire knowledge bases opposed to our works that quantify inconsistent at the level of tuple. We quantify inconsistency for each tuple in database and for each query answer.

3.1.2 Inconsistency in relational database

Concerning the inconsistency problem in relational databases, it has been mainly tackled by the database repair approaches; also, the consistent query answering approach (CQA) based on these database repair approaches is developed to answer queries on inconsistent databases [6, 8, 19–21, 32, 58, 59, 64, 72, 98, 103, 104, 106, 107, 111, 119, 120, 120, 146, 146, 150, 155]. The repairing approach consists to restore the database consistency by making deletion of tuples from database, or addition of new tuples in database, or modification of values of some attributes, etc [20–22]. These actions to restore the database consistency are called semantic of repairs [22]. A database in which the consistency is restored is called repair. Each semantic of repair can generate many repairs (possible infinite set of repairs) [22]. Given a query Q and a semantic of repair Sem, the consistent query answers is the intersection of answers of Q computed over all the repairs obtained with the semantic Sem [22].

The repairing and CQA notion have been first introduced by Arena et al [9], and in this seminal paper they propose repair by deletion. The main semantics of repairs are: repair by deletion [9, 32, 106, 146, 150] consisting to restore consisting by deleting some tuples, repair by insertion[120] that restores consistency by inserting (possibly fictive) tuples and repair by update [19, 72, 104, 107, 111, 120] that modifies values of attributes to restore consistency.

The theoretical complexity of CQA have been largely investigated in literature [31, 33–35, 52, 60, 75, 92, 92, 93, 95–97, 110, 118, 120, 147–149, 151], these research works have shown that computing CQA is a Co-NP complete problem. Other works are focused on specific classes of queries and constraints. So, they have proposed for these classes a rewriting approach that enables to compute CQA in polynomial time in data complexity [61–63]. Also, in the same direction to find efficient algorithm to compute CQA, some quantitative approaches that quantify inconsistency are defined [97, 105, 108, 117], but all these works depend to a repairing approach. For more details about repairing and CQA, the reader can be referred to the Bertossi's book [22].

Several repairs approaches have been defined in literature [22]. Depending to the repair approach, the consistent query answers can be different from a repair approach to an other and CQA keeps only a subset of answers of the query [9]. Another problem of the repairing and CQA approach is the fact that the number of repairs generated is exponential (can be infinite) in the size of the database; and in the best case, the complexity of CQA is CO-NP-complete [148]. Opposed to our work, we propose a set of measures that quantify inconsistency and that do not assume any repairing approach. Also, we allow a ranking of query answers according to the their inconsistency degrees.

3.2 Top-k algorithms

In this section, we discuss the works related to top-k queries. Two classes of algorithms have been considered: Disk-based algorithms and Memory-based algorithms.

3.2.1 Disk-based algorithms

In this class of top-k algorithms, the top-k answers are computed while optimizing the input/output operations (i.e, read and write operations made on disk). A seminal work in this area is developed by Fagin [51] through the famous FA(Fagin algorithm) instance optimal algorithm [53]. Given m lists of items and each item has a score (the lists contains the same items with not necessary the same scores), the FA algorithm sorts first each list and in each one it takes the k first items with the high scores; in second step, it aggregates the scores of items (by an aggregate monotone scoring function) from different lists and when an item is not loaded in one or more of the lists then it is accessed by a random access. The FA is extended to TA algorithm by integration of notion of scoring bound [53]. The TA algorithm works exactly as the FA algorithm and assumes the same conditions as FA but the items in lists are sequentially accessed one after one and a threshold score is computed after each read of item; this threshold score enables to know what item, among seen items, belongs to the top-k. The algorithms FA and TA assumes a random access and a sequential access; NRA [53] is developed to work only with sequential access. Similar works have been investigated in [29, 76, 115]. The above top-k algorithms assume a simple input selection query with only natural join on key attributes. Then, Natsev et al [125] introduce the J^* algorithm that works with the general join, J^* assumes only the sequential access. Other top-k algorithms for general join are proposed in [77, 83, 84, 101, 130, 131], all these works assume a sequential access (with sorted relations). In [102] the authors propose a top-k algorithm for queries with aggregation clause and/or aggregation function. A survey of the top-k algorithms is presented in [85].

These research works have developed top-k algorithms to work only for monotonic functions and the input data are, at least, sequentially accessed in sorting order of scores. In this thesis, we design top-k algorithms for non-monotonic scoring functions.

3.2.2 Memory-based algorithms

The class of Memory-based algorithms does not take into account minimization of input/output operations, it loads all the database in main memory and computes top-k answers while minimizing operations in main memory.

The referential work in this area is investigated by Tziavel et al in [140]. The authors combine the new techniques developed to efficiently compute the join [65, 66, 157]

and ranking enumeration while assuming to have monotonic functions. These efficient techniques generate first the join tree of the query, and based to the properties of this join tree [66], an efficient technique can be used to compute join results. So, in [140], the authors fusion efficient join and ranking to enumerate the k answers of queries in order of their scores (where the aggregate scoring function is the sum). First they perform for path queries (where the join tree of the query can be converted into a path), so they convert the problem of finding the k answers enumeration into a problem of finding the k shortest paths. They generalize beyond path queries. For an acyclic query, one of their algorithms computes the top-k in $O(n + k * log(k))$ in data complexity with n the size of data. The authors of [48] propose a recursive version of enumeration of answers of queries in order of their scores, their approach works also with monotonic functions, for an acyclic query it runs in $O(n + k * log(n))$. Other top-k algorithms, in this class, are also proposed in [26, 30, 39, 78, 160].

Aside from algorithm developed in [160], that is designed for generic function (not necessary a monotonic function), all these algorithms work with monotonic aggregate scoring function. In addition, these algorithms do not work for queries with aggregation clause. In this thesis, we develop top-k algorithm for some aggregate queries (with MIN and MAX functions) and for non monotonic functions.

3.3 Conclusion

We presented in this chapter, the state of art. We divided this start of art into four parts. The first part described works concerning the handling of inconsistency in knowledge bases; the second one was about inconsistency problem fixing in relational database, mainly it was consecrated to the database repairing and consistent query answering notions; the third part was dedicated to the top-k algorithms in Disk-based algorithms class(algorithms that minimize the input/output operations on disk); and the last part have presented the top-k algorithms in Memory-based algorithms (i.e, those algorithms that load all data in main memory and minimize the number of operations in main memory).

In the next chapter, we present one of our main works that is about the introduction of a set of measures of inconsistency degrees.

Chapter 4

QUANTIFYING INCONSISTENCY

In this chapter, we propose a set of measures to quantify inconsistency degrees in relational databases. We focus on the problem of quantifying the inconsistency degrees of query answers.

A set of constraints SC is consistent if only if there exists a non empty database D such that $D \models SC$. We assume that the considered set of constraints, in this chapter, is consistent. This assumption will be lifted in chapter 6. The proposed measures enable to quantify the inconsistency degrees of a given tuple that can be a base tuple or a query answer. We introduce two classes of measures: tuple-based measures and constraint-based measures that we describe below in detail.

Let $\mathcal{I}$ be a database instance over a schema $\mathcal{S}$ and let DC be a set of denial constraints over $\mathcal{S}$. We proceed in three steps in order to define the set of measures of inconsistency degrees.

- *Identifying inconsistent tuples:* We first start by identifying the inconsistent tuples of an instance $\mathcal{I}$ over $\mathcal{S}$ with respect to a set of denials constraints DC. To achieve this task, we turn each denial constraint $C \in DC$ into a Boolean conjunctive query denoted Q^C. We use the why-provenance (also known as lineage) of each Q^C to compute the set of inconsistent tuples in $\mathcal{I}$ with respect to DC. The set of inconsistent tuples of an instance $\mathcal{I}$ with respect to a set of denial constraints DC is denoted by $IncT(\mathcal{I}, DC)$.

- *Annotating the initial database instance:* Using the why-provenance of each Boolean conjunctive query obtained from each denial constraint, we convert the instance $\mathcal{I}$ into a *K-instance* by annotating each consistent tuple in $\mathcal{I}$ with the value 1 and each inconsistent tuple $t \in IncT(\mathcal{I}, DC)$ with $id(t) \times C_{1_{id}} \times \cdots \times C_{p_{id}}$, where $C_1, \ldots, C_p \in DC$ and t appears in why-provenance of C_i with $i \in [1, n]$.

- *Defining inconsistency degrees of query answers:* then, given an **UCQ** Q over the instance $\mathcal{I}$, we use polynomial provenance semi-ring to annotate the query answers. The latter provenance is the most informative form of provenance annotation [73] and hence is exploited in our setting in order to define different inconsistency degrees for query answers.

In the next section, we illustrate our approach to quantify inconsistency degrees.

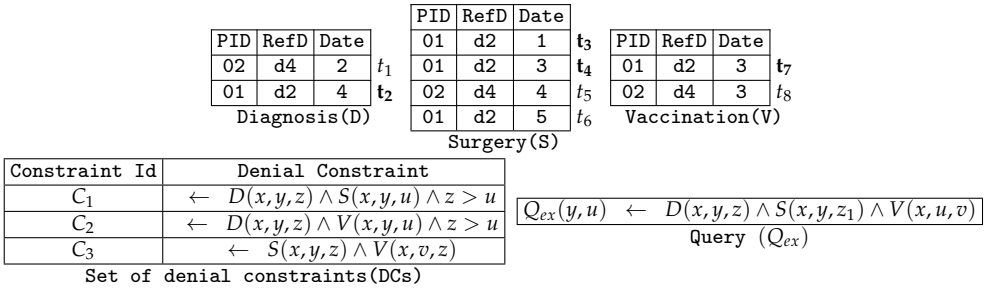


Figure 4.1: A hospital database hdb with a set of denial constraints (DCs) and a query Q_{ex}.

Answers	Contrib. tuples	Violated Constr.	# Constr. Violations	
$\langle d2, d2\rangle$	$\mathbf{t_2}$, $\mathbf{t_3}$, $\mathbf{t_7}$	$C_1 \times C_2 \times C_3$	$C_1^2 \times C_2^2 \times C_3$	a_1
$\langle d2, d2\rangle$	$\mathbf{t_2}$, $\mathbf{t_4}$, $\mathbf{t_7}$	$C_1 \times C_2 \times C_3$	$C_1^2 \times C_2^2 \times C_3^2$	a_2
$\langle d2, d2\rangle$	$\{\mathbf{t_2}, t_6, \mathbf{t_7}\}$	$C_1 \times C_2 \times C_3$	$C_1 \times C_2^2 \times C_3$	a_3
$\langle d4, d4\rangle$	$\{t_1, t_5, t_8\}$	1	1	a_4

Table 4.1: Annotated answers of query Q_{ex}

4.1 Illustrative Example

Consider a relational database instance $\mathcal{I}$ in Figure 4.1 consisting three relations D, V and S with a corresponding number of denial constraints IC and a query Q_{ex}. In each relation in $\mathcal{I}$, the first column is the patient identifier *PID*, the second column is the desease reference *RefID* and the third column is the *Date* of a given event. Notice that the schema of the three tables is the same solely for illustration purposes and to maximize the number of joins across the tables. In fact, our methods are generalizing to relations with an arbitrary schema. The denial constraint (C_1) imposes to have any diagnosis for a patient's disease before surgery for the same disease concerning the same patient. The constraint (C_2) and (C_3) establish that a patient cannot be diagnosed a given disease for which he/she has been administered a vaccine on a previous date[1]. Finally, a conjunctive query Q_{ex} extracts pairs of diseases for which the same patient underwent surgey and was administered a vaccine. The tuples in the relations highlighted in red are those that violate one or more constraints (C_1), (C_2) or (C_3).

Before evaluating the query Q_{ex}, we annotate each tuple in the databases instance I with an unique identifier. When applying the why-provenance to the tuples augmented with their identifiers, and for each answer tuple of query Q_{ex}, the corresponding possible derivations in terms of tuple identifiers are shown (see Table 4.1). We can notice that each answer tuple contains bold tuple tuple identifiers. By counting the number of bold identifiers, we can compute the inconsistency degree of the tuple. For instance, the answer tuple $\{R_1, R_2\}$ has inconsistency degree equal to 3, whereas the answer tuple $\{R_7, R_4\}$ will have inconsistency degree equal to 3 or 2, depending on whether we favor a greater or smaller number of derivating inconsistent tuples, and so on. We can also obtain duplicates in the provenance column that might be taken into account in the counting or not. These considerations led us to precisely define twelve measures of inconsistency degrees under set- and bag-semantics and to provide prac-

[1]Notice that there are exceptions to the last two constraints when a second shot of a vaccine is somministrated or when the immunization offered by a vaccine did not work properly. These cases would be covered by associating probabilities to the constraints, which we do not consider in our work for the time being.

tical methods to compute them starting from plain database instances. We tackle these questions in the remainder of the thesis.

In the next section, we shall detail the proposed three-step approach.

4.2 Identifying inconsistent tuples

Let $\mathcal{I}$ be an instance over a database schema $\mathcal{S}$ and let DC be a set of denial contraints over $\mathcal{S}$. We first convert the set denial constraints DC into a set of boolean conjunctive queries denoted Q^{DC} as follows. For each constraint $C \in DC$ of the form:

$$\leftarrow R_1(u_1) \wedge ... \wedge R_n(u_n) \wedge \phi(u_1, \ldots, u_n)$$

we generate a boolean conjunctive query Q^C:

$$Q^C() \leftarrow R_1(u_1) \wedge ... \wedge R_n(u_n) \wedge \phi(u_1, \ldots, u_n)$$

Example 4.1. *The set DC of denial constraints depicted in Figure 4.1 leads to the following set of boolean conjunctive queries:*

$Q^{C_1}() \leftarrow D(x,y,z) \wedge S(x,y,u) \wedge z > u$
$Q^{C_2}() \leftarrow D(x,'R2',y) \wedge V(x,'R2',z) \wedge y > z$
$Q^{C_3}() \leftarrow D(x,'R4',y) \wedge V(x,'R4',z) \wedge y > z$

It is easy to verify that an instance $\mathcal{I}$ violates the set of denial constraints DC iff at least on boolean conjunctive query from Q^{DC} evaluates to true over the instance $\mathcal{I}$ (i.e., $\exists C \in DC, Q^C(\mathcal{I}) = \{<>\}$, where the empty tuple <> denotes the *true* value of a boolean query). The lineage of the empty tuple <> enables the identification of the set of all contributing source tuples in violation, and hence all the tuples that *"contribute"* to make the instance $\mathcal{I}$ inconsistent w.r.t. DC. We shall use the *provenance semirings* [73] to compute it.

Let $\mathcal{P}(\Gamma)$ be the powerset of the set of tuple identifiers Γ. Consider the following provenance semiring: $(\mathcal{P}(\Gamma) \cup \{\bot\}, +, ., \bot, \emptyset)$, where $\forall S, T \in \mathcal{P}(\Gamma) \cup \{\bot\}$, we have $\bot + S = S + \bot = S, \bot.S = S.\bot = \bot$ and $S + T = S.T = S \cup T$ if $S \neq \bot$ and $T \neq \bot$. This semiring consists of the powerset of Γ augmented with the distinguished element $\bot$ and equipped with the set union operation which is used both as addition and multiplication. The distinguished element $\bot$ is the neutral element of the addition and the annihilating element of the multiplication.

We convert the instance $\mathcal{I}$ over the schema $\mathcal{S}$ into a *K-instance*, denoted by $\mathcal{I}^{LP}$, with $K = \mathcal{P}(\Gamma) \cup \{\bot\}$. The *K-instance* $\mathcal{I}^{LP}$ is defined below.

Definition 4.1 (Lineage provenance database)**.** *Let $\mathcal{I}$ be an instance over a database schema $\mathcal{S}$ and let DC be a set of denial contraints over $\mathcal{S}$. Let $K = \mathcal{P}(\Gamma) \cup \{\bot\}$. The K-instance $\mathcal{I}^{LP}$, called lineage provenance database from $\mathcal{I}$, is constructed as follows:*

- *$\forall R_i \in S$ a corresponding K-relation is created in $\mathcal{I}^{LP}$,*
- *A K-relation $\mathcal{I}^{LP}(R_i) \in \mathcal{I}^{LP}$ is populated as follows:*

$$\begin{cases} \mathcal{I}^{LP}(R_i)(t) = \{id(t)\} & \text{if } t \in \mathcal{I}(R_i) \\ \mathcal{I}^{LP}(R_i)(t) = \bot & \text{otherwise} \end{cases}$$

Example 4.2. *Figure 4.2 shows the lineage provenance database hdb^{LP} obtained from the hospital database hdb by annotating each tuple $t \in \mathsf{hdb}$ with a singleton set $\{id(t)\}$ containing the tuple identifier. The column **lprov** contains the annotation of each tuple.*

PID	RefD	Date	lprov
02	$d4$	2	$\{t_1\}$
01	$d2$	4	$\{t_2\}$

Diagnosis (D)

PID	RefD	Date	lprov
01	$d2$	1	$\{t_3\}$
01	$d2$	3	$\{t_4\}$
02	$d4$	4	$\{t_5\}$
01	$d2$	5	$\{t_6\}$

Surgery (S)

PID	RefD	Date	lprov
01	$d2$	3	$\{t_7\}$
02	$d4$	3	$\{t_8\}$

Vaccination (V)

Figure 4.2: The Lineage provenance databases hdb^{LP} obtained from the hospital database hdb

Using the provenance semirings, we define below the inconsistent tuples of a given instance w.r.t. a set of denial constraints.

Definition 4.2 (Inconsistent tuples)**.** *Given an instance I and a set of denial constraints DC, the set of inconsistent tuple identifiers, denoted by $IncT(\mathcal{I}, DC)$, is defined as follows*

$$IncT(\mathcal{I}, DC) \stackrel{def}{=} \bigcup_{Q \in Q^{DC}} Q((\mathcal{I})^{LP})(\langle\rangle)$$

Consequently, a tuple $t \in \mathcal{I}$ is inconsistent w.r.t. DC if $id(t) \in IncT(\mathcal{I}, DC)$.

In the following, we define the notion of violated constraints by a tuple from input database.

Definition 4.3 (Violated constraints)**.** *Given an instance $\mathcal{I}$ and a set of denial constraints DC, the set $VC(\mathcal{I}, DC, t)$ of constraints of DC violated by a tuple $t \in \mathcal{I}$ is defined as follows: $VC(\mathcal{I}, DC, t) = \{C \in \mathrm{Y} \mid t \in Q^C(\mathcal{I}^{LP})(<>)\}$*

Example 4.3. *Consider the set of boolean conjunctive queries Q^{DC} of Example 4.1 which is obtained from the set of denial constraints of the hdb database. The execution of each query Q^C from Q^{DC} over the lineage provenance database hdb^{LP} of Figure 4.3 leads to the answer true (i.e., the tuple <>), if it is violated, annotated with the set of tuples from database hdb that involve in violation of C, i.e.,:*

- $Q^{C_1}(\mathsf{hdb}^{LP})(<>) = \{t_2, t_3, t_4\}$
- $Q^{C_2}(\mathsf{hdb}^{LP})(<>) = \{t_2, t_7\}$
- $Q^{C_3}(\mathsf{hdb}^{LP})(<>) = \{t_4, t_7\}$

So, the set of inconsistent tuples is the following:

$$incT(\mathsf{hdb}, DC) = \{t_1,\ t_{17},\ t_2,\ t_{18},\ t_3,\ t_{19},\ t_7,\ t_{10},\ t_6,\ t_{11},\ t_5,\ t_{13}\}$$

The violated constraints by tuple t_2 are $VC(\mathcal{I}, DC, t_1) = \{C_1,\ C_2\}$, that are violated by t_7 are $VC(\mathcal{I}, DC, t_7) = \{C_2,\ C_3\}$.

In the next section, we show how the final instance is annotated.

4.3 Annotation of Database

In the next section, we will show how to use the provenance polynomials to define the inconsistency degrees of query answers. To achieve this task, we first need to convert the instance $\mathcal{I}$ into a $\mathbb{N}[\mathrm{Y} \cup \Gamma]$-*instance*, denoted $\mathcal{I}^{\mathrm{Y} \cup \Gamma}$. As shown in the following definition, an instance $\mathcal{I}^{\mathrm{Y} \cup \Gamma}$ is derived from $\mathcal{I}$ by tagging each tuple $t \in \mathcal{I}$ with a monomial with variables in $\mathrm{Y} \cup \Gamma$. These monomials, during query evaluation, are propagated to query answers thus enabling to identify the inconsistent tuples used to computed a given answer as well as the constraints violated by this last one.

PID	RefD	Date	prov
02	$d4$	2	1
01	$d2$	4	$t_2C_1C_2$

Diagnosis (D)

PID	RefD	Date	prov
01	$d2$	1	t_3C_1
01	$d2$	3	$t_4C_1C_3$
02	$d4$	4	1
01	$d2$	5	1

Surgery (S)

PID	RefD	Date	prov
01	$d2$	3	$t_7C_2C_3$
02	$d4$	3	1

Vaccination (V)

Figure 4.3: The K-instances hdb^{LP} (without prov column) and hdb^{Υ} (without lprov column).

Definition 4.4 ($\mathcal{I}^{\Upsilon\cup\Gamma}$ instance)**.** *Let $\mathcal{I}$ be an instance over a database schema $\mathcal{S}$ and let DC be a set of denial constraints over $\mathcal{S}$. Let $K = \mathbb{N}[\Upsilon \cup \Gamma]$. The K-instance $\mathcal{I}^{\Upsilon\cup\Gamma}$ is constructed as follows: $\forall R_i \in S$ a corresponding K-relation is created in $\mathcal{I}^{\Upsilon\cup\Gamma}$. A K-relation $\mathcal{I}^{\Upsilon\cup\Gamma}(R_i) \in \mathcal{I}^{\Upsilon\cup\Gamma}$ is populated as follows:*

$$\mathcal{I}^{\Upsilon\cup\Gamma}(R_i)(t) = \begin{cases} 0 & if\ t \notin \mathcal{I}^{\Upsilon}(R_i) \\ t_{id}^{d} \times \prod\limits_{C_{id}\in\Upsilon} C_{id}^{l} & otherwise \end{cases}$$

with $l = 1$ if $C_{id} \in VC(\mathcal{I}, DC, t)$ or $l = 0$ otherwise; and t_{id} is identifier of t considered as a variable and $d = 1$ if $t \in IncT(\mathcal{I}, DC)$ or $d = 0$ otherwise.

Hence, an annotation $\mathcal{I}^{\Upsilon\cup\Gamma}(R_i)(t)$ of a tuple t is equal to 1 if the base tuple t is consistent (i.e., $VC(\mathcal{I}, DC, t) = \emptyset$), otherwise it is equal to a monomial expression that uses as variables the identifiers of the constraints violated by t (i.e., the elements of $VC(\mathcal{I}, DC, t)$).

Example 4.4. *Continuing with our example, the hdb^{Υ} instance obtained from the hospital database hdb is depicted in Figure 4.3. We illustrate below the computation of the annotations of the tuples t_1 (a consistent tuple) and t_2 (an inconsistent tuple). From the previous example, we have $VC(\mathcal{I}, DC, t_1) = \emptyset$ and hence the annotation of t_1 is computed as follows: $hdb^{\Upsilon}(R_i)(t_1) = 1$ because $t_1 \notin IncT(\mathcal{I}, DC)$. For the tuple t_2, we have $VC(\mathcal{I}, DC, t_2) = \{C_1, C_2\}$ and hence: $hdb^{\Upsilon}(R_i)(t_2) = t_2 \times C_1^1 \times C_2^1 \times C_3^0 = t_2C_1C_2$.*

Constraint-based formula, denoted $Cs(P) : \mathbb{N}[\Upsilon \cup \Gamma] \to \mathbb{N}[\Upsilon]$ is a function that takes as input a polynomial of variables from $\Upsilon \cup \Gamma$ and returns a polynomial of variables from Υ where each variable from Γ takes value 1.
Tuple-based formula, denoted $Tp(P) : \mathbb{N}[\Upsilon \cup \Gamma] \to \mathbb{N}[\Gamma]$ is a function that takes as input a polynomial of variables from $\Upsilon \cup \Gamma$ and returns a polynomial of variables from Γ where each variable from Υ takes value 1.

Example 4.5. *Consider $\Gamma = \{t_1, t_2, t_3, t_4, t_5\}$ and $\Upsilon = \{c_1, c_2, c_3, c_4\}$. Let $P = t_1^2c_1t_3 + c_3c_4c_2 + t_1t_2$ be a polynomial of variables from $\Gamma \cup \Upsilon$, then*

- $Cs(P) = c_1 + c_3c_4c_2 + 1$
- $Tp(P) = t_1^2t_3 + 1 + t_1t_2$

In the sequel, we assume that the relations of an annotated instance $\mathcal{I}^{\Upsilon\cup\Gamma}$ are augmented with an attribute *prov* that stores the annotations of the base tuples. As an example, Figure 4.3 shows the annotated relations of the instance $hdb^{\Upsilon\cup\Gamma}$ together with their respective *prov* columns.

Now, it is ready to define the measures of inconsistency degrees. In the next section, we define the twelve measures of inconsistency degrees (shown in figure 4.4).

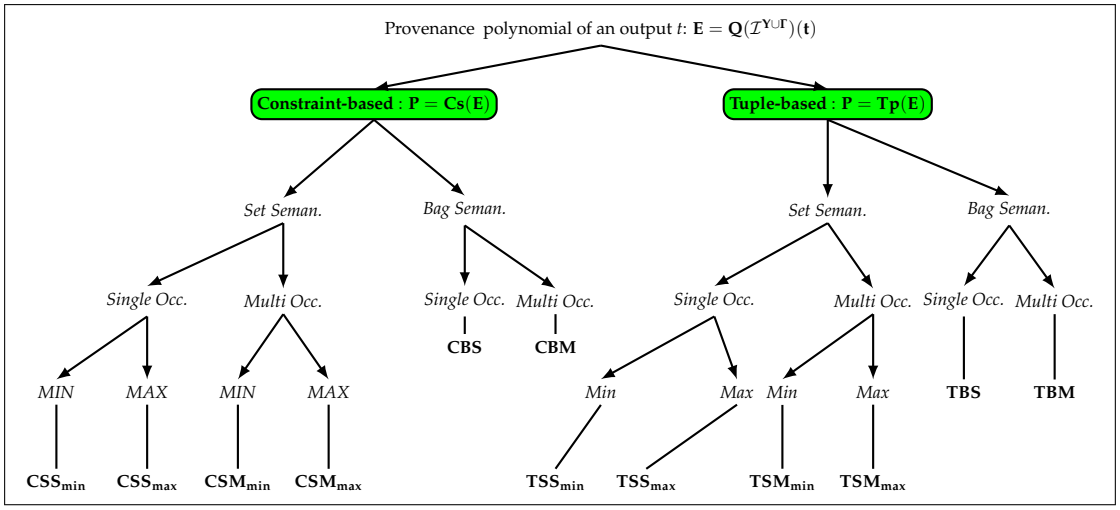


Figure 4.4: The various quantification of inconsistency that can be obtained along with their criteria.

4.4 Query answers inconsistency

Given a query Q, we evaluate Q over the $\mathbb{N}[\mathrm{Y}\cup\Gamma]$-instance $\mathcal{I}^{\mathrm{Y}\cup\Gamma}$ and use the provenance polynomials semiring in order to annotate each answer t of Q. The computed annotations, expressed as polynomials with variables from the set $\mathrm{Y}\cup\Gamma$ consisting of the identifiers of tuples and identifiers of constraints, are fairly informative as they allow to fully record how the constraints are violated by base tuples that contribute to each answer and how inconsistent tuples are used to compute these answers. Such annotations are hence exploited to compute the various inconsistency measures needed for query answers.

Example 4.6. *Continuing with the example, evaluating the query Q_{ex} over $hdb^{\mathrm{Y}\cup\Gamma}$ and computing its polynomial provenance leads to the following annotated answers:*
$Q_{ex}(hdb^{\mathrm{Y}})(\langle d2, d2\rangle) = t_2t_3t_7C_1^2C_2^2C_3 + t_2t_4t_7C_1^2C_2^2C_3^2 + t_2t_7C_1C_2^2C_3$
$Q_{ex}(hdb^{\mathrm{Y}})(\langle d4, d4\rangle) = 1.$

The monomial $t_2t_3t_7C_1^2C_2^2C_3$ that appears in the annotation of the answer $\langle d2, d2\rangle$ of Q_{ex} encodes the fact that this answer can be computed from inconsistent base tuples t_2, t_3, t_7 that lead to the violation of the constraints C_1 and C_2 twice and to the violation of the constraint C_3 once.

Hence, the polynomial expression $Q(\mathcal{I}^{\mathrm{Y}\cup\Gamma})(t)$ fully records the inconsistency of an output t in terms of violations of constraints, the inconsistent tuples used to compute it and therefore can be used to quantify the inconsistency degrees of a query outputs. Consider a polynomial $P = Q(\mathcal{I}^{\mathrm{Y}\cup\Gamma})(t)$ of an output t of a given query Q. Each monomial M from P gives an alternative way to derive the output t.

Let $\mathcal{I}, Q, DC$ be an instance database, an union of conjunctive queries, a set of denial constraints with identifiers the set Y, respectively. It is straightforward to notice that in bag semantic answers, $\forall t \in Q(\mathcal{I})$ then $Q(\mathcal{I}^{\mathrm{Y}\cup\Gamma})(t)$ is a monomial.

We categorize the inconsistency measures into two classes:
Tuple-based measures, which quantify inconsistency by using exclusively inconsistent tuples.

Constraint-based measures, the measures in this class quantify inconsistency using exclusively violated constraints.

These measures of inconsistency are described below.

4.4.1 Definition of measures

Given an **UCQ** query Q, we evaluate Q over the instance $\mathcal{I}^{\Upsilon \cup \Gamma}$ in order to compute the answers of Q as well as the provenance polynomials semiring annotations associated with each answer. The annotations, which come in the form of polynomial expressions, are then exploited to define several inconsistency measures for query answers.

Let $\mathcal{I}$, Q and DC be, respectively, an instance, an **UCQ** query and a set of denial constraints over a database schema $\mathcal{S}$. Let $t \in Q(\mathcal{I})$ be an answer of the query Q over the instance $\mathcal{I}$. Let $K = \mathbb{N}[\Upsilon \cup \Gamma]$. Applying the query Q to the *K-instance* $\mathcal{I}^{\Upsilon \cup \Gamma}$, enables to compute the provenance annotation $Q(\mathcal{I}^{\Upsilon \cup \Gamma})(t) = P$ associated with each answer $t \in Q(\mathcal{I})$. This tuple based annotation consists in a polynomial expression P over the set of variables Γ. Recall that the variables that appear in P correspond to identifiers of inconsistent tuples (i.e., elements of $IncT(\mathcal{I}, DC)$). Hence, the polynomial P fully documents how inconsistent source tuples contribute in the computation of the output t. This polynomial also encodes the violated constraints and their occurrence violations. In particular, each monomial $M \in M(P)$ gives an alternative way to compute the output t. Based on the polynomial annotation P of a tuple t, different measures can be defined in order to quantify the inconsistency degree of t depending in particular on how one deals with the following three issues:

- Considering query answers semantics, either bag semantics or set semantics, different measures can be developed depending on query answers semantics. When set semantics of query answers is considered, a query answer is computed in several alternative ways. Opposite to the case of bag semantic where a query answer is compute from only one alternative. So, in the case of set semantics, the measures defined consider all the alternative ways while measures in the case of bag semantics have to consider only one alternative way.

- How to deal with a base tuple that contribute more than one time in the computation of the same query answer t? We define two cases: *single occurrence*, in which a contribution of a source tuple in the computation of an answer is counted at most once, and *multiple occurrence*, where the exact number of contributions of a source tuple is taken into account when quantifying the inconsistency degree of a given answer.

- What kind of measures to consider? Either tuple-based measures or constraint-based measures.

The combination of the previous dimensions leads to the following twelve measures of inconsistency, as reported in table 4.2. In the rest of this chapter, we assume that t is an answer of Q from a database instance $\mathcal{I}$ in presence of a set of denial constraints DC and $P = Q(\mathcal{I}^{\Upsilon \cup \Gamma})(t)$. We assume the following notations: $T^p = Tp(P)$ and $C^s = Cs(P)$.

- *Set semantic of query answers*. In this semantic, as a given answer t is computed from different ways then $|M(P)| \geq 1$, i.e the provenance polynomial of t is a polynomial composed from different monomials.

– *Single occurrence*, in this semantics, the quantification of an inconsistency degree from a monomial $M \in M(P)$ is achieved by counting the number of variables in M (since duplicate contributions are not counted, the exponents of the variables are dropped). The set semantics leads to two measures depending on how alternatives are dealt with:
  * MIN alternative: is the alternative that considers the way to compute the answer with a minimum inconsistency. Depending to the class of measures, we have the following two measures:
    · *Tuple-based measure*: this measure allows to compute the given answer with the possible minimum number of inconsistent tuples. Hereafter, this measure is denoted by TSS_{min} and it is formally defined as follows:

$$TSS_{min}(t, \mathcal{I}, Q, DC) = \min_{M \in M(T^p)} Var(M)$$

    · *Constraint-based measure*: this measure quantifies inconsistency by considering the alternative way to compute a given answer containing the minimum possible violated constraints. Hereafter, this measure is denoted by CSS_{min}. It is formally defined as follows:

$$CSS_{min}(t, \mathcal{I}, Q, DC) = \min_{M \in M(C_s)} Var(M)$$

Example 4.7. *Continuing our running example, the valuation of the query Q_{ex} of Figure 4.1 is processed as illustrated in Table 4.5. Consider the annotation* $Q_{ex}(hdb^{\mathrm{Y}})(\langle d2, d2\rangle) = \underbrace{C_1^2C_2^2C_3t_2t_3t_7}_{M_1} + \underbrace{C_1^2C_2^2C_3^2t_2t_4t_7}_{M_2} + \underbrace{C_1C_2^2C_3t_2t_7}_{M_3}$.
This annotation conveys the information about the violated constraints by each of the three possible ways to derive the output $\langle d2, d2\rangle$ as an answer to the query Q_{ex}. It shows also the inconsistent tuples used to compute the output $\langle d2, d2\rangle$, also the violated constraints. We have $T^p = t_2t_3t_7 + t_2t_4t_7 + t_7$ and $C_s = C_1^2C_2^2C_3 + C_1^2C_2^2C_3^2 + C_1C_2^2C_3$, and $TSS_{min}(\langle d2, d2\rangle, \mathcal{I}, Q, DC) = 1$ $CSS_{min}(\langle d2, d2\rangle, \mathcal{I}, Q, DC) = 3$ since we have only t_7 that is necessary as inconsistent tuple to compute $\langle d2, d2\rangle$ and C_1, C_2, C_3 are violated.
  * MAX alternative: this alternative considers the way to compute the answer with a maximum inconsistency. Considering the class of measures, we have the following measures:
    · *Tuple-based measure*: this measure takes the alternative way to compute answer with the maximum number number of inconsistent tuples. We denote this measure by TSS_{max} and it is formally defined as follows.

$$TSS_{max}(t, \mathcal{I}, Q, DC) = \max_{M \in M(T^p)} Var(M)$$

    · *Constraint-based measure*: it considers the alternative way to compute a given answer with the possibly large number of violated constraints. We denote this measure by CSS_{max}. It is formally defined as follows

$$CSS_{max}(t, \mathcal{I}, Q, DC) = \max_{M \in M(C_s)} Var(M)$$

Example 4.8. *Considering example 4.7, we have* $TSS_{max}(\langle d2, d2\rangle, \mathcal{I}, Q, DC) = 3$ *and* $CSS_{max}(\langle d2, d2\rangle, \mathcal{I}, Q, DC) = 3$ *since the maximal number of inconsistent tuples used is* 3 (t_2, t_3, t_7 *or* t_2, t_4, t_7*) and* $C_1, C_2, C3$ *are violated.*

– *Multiple occurrence*. In this semantics, the quantification of an inconsistency degree from a monomial $M \in M(P)$ is achieved by computing the weight of M. Hence, if an inconsistent source tuple is used n times in the monomial M or a constraint is violated n times in the monomial M, it will then be counted n times in the quantification of the inconsistency degree from M. The multiple occurrence leads to four measures depending on how alternatives are dealt with:

  * The MIN alternative shows the option of minimum inconsistency while considering multiple occurrence. Two measures can be defined depending to the classes of measures.
    · *Tuple-based measure*: counts the number of inconsistent tuples used to compute the given answer with the minimum number of use of inconsistent tuples. Denoted by TSM_{min}, it is formally defined as follows:

$$TSM_{min}(t, \mathcal{I}, Q, DC) = \min_{M \in M(T^p)} W(M)$$

    · *Constraint-based measure*: takes as inconsistent degrees the number of times the constraints are violated in order to compute the given answer when the minimum number of times constraints are violated. Denoted by CSM_{min}, it is formally defined as follows:

$$CSM_{min}(t, \mathcal{I}, Q, DC) = \min_{M \in M(C_s)} W(M)$$

  Example 4.9. *Considering example 4.7, we have* $TSM_{min}(\langle d2, d2\rangle, \mathcal{I}, Q, DC) = 1$ *and* $CSM_{min}(\langle d2, d2\rangle, \mathcal{I}, Q, DC) = 4$ *since the at least* t_7 *is used to compute answer and* $C_1, C_2, C3$ *are violated one time, two times, one time respectively.*

  * The MAX alternative corresponds to the pessimist option opposite to Min alternative, i.e, in this option, we consider the maximum number of violations of constraints or the maximum use of inconsistent source tuples contribute in computation of an answer.
    · *Tuple based approach*: counts the maximum number of times inconsistent tuples are used in different derivations to compute the given answer, this measure is denoted TSM_{max} and is defined as follows

$$TSM_{max}(t, \mathcal{I}, Q, DC) = \max_{M \in M(T^p)} W(M)$$

    · *Constraint based approach*: counts the maximum number of times the constraints are violated among alternative ways to compute the given answer, it is called CSM_{max} and it is defined as follows

$$CSM_{max}(t, \mathcal{I}, Q, DC) = \max_{M \in M(C_s)} W(M)$$

  Example 4.10. *Considering example 4.7, we have* $TSM_{max}(\langle d2, d2\rangle, \mathcal{I}, Q, DC) = 3$ *and* $CSM_{max}(\langle d2, d2\rangle, \mathcal{I}, Q, DC) = 6$ *since the at most* t_2, t_3, t_7 *are used to compute answer and* $C_1, C_2, C3$ *are violated two time, two times, two times respectively.*

	Tuple-based measures		Constraint-based measures	
	Multi-occurrence	**Single-occurrence**	**Multi-occurrence**	**Single-occurrence**
Set Sem.	$TSM_{min} \equiv \min_{M \in M(T^p)} W(M)$	$TSS_{min} \equiv \min_{M \in M(T^p)} Var(M)$	$CSM_{min} \equiv \min_{M \in M(C_s)} W(M)$	$CSS_{min} \equiv \min_{M \in M(C_s)} Var(M)$
	$TSM_{max} \equiv \max_{M \in M(T^p)} W(M)$	$TSS_{max} \equiv \max_{M \in M(T^p)} Var(M)$	$CSM_{max} \equiv \max_{M \in M(C_s)} W(M)$	$CSS_{max} \equiv \max_{M \in M(C_s)} Var(M)$
Bag Sem.	$TBM \equiv W(T^p)$	$TBS \equiv Var(T^p)$	$CBM \equiv W(C_s)$	$CBS \equiv Var(C_s)$

Table 4.2: Different proposed measures of inconsistency with their formal definitions. The notation $\alpha \equiv \beta$ means $\alpha(t, \mathcal{I}, Q, DC) = \beta$ with α a measure of inconsistency and β its definition

- *Bag semantic of query answers*, in this semantics, each answer is computed in one processing manner. It corresponds to the semantics of **SQL** language. For this semantic, given an answer t of an union of conjunctive queries Q evaluated over an instance $\mathcal{I}$ in the presence of a set of denial constraints DC, $Q(\mathcal{I}^{\Upsilon \cup \Gamma})(t)$ is a monomial, so P is a monomial.
  - *Single occurrence*. In this case, we count one time a violation, either constraints violations or violations of base tuples.
    - *Tuple based approach*, counts the number of inconsistent tuples used to compute answer. It is called TBS and it defined as follows

      $$TBS(t, \mathcal{I}, Q, DC) = Var(T^p)$$

    - *Constraint based approach*, counts the number of constraints violated by base tuples used to compute answer. It is denoted CBS.

      $$CBS(t, \mathcal{I}, Q, DC) = Var(C_s)$$

  - *Multiple occurrence*. We count exactly the number of violations, either constraints violations or violations of base tuples.
    - *Tuple based approach*, called TBM, counts the number of inconsistent tuples (with repetition) used in computation of answer.

      $$TBM(t, \mathcal{I}, Q, DC) = W(T^p)$$

    - *Constraint based approach*, it counts the number of violated constraints, with repetition, by answer. It is called CBM.

      $$CBM(t, \mathcal{I}, Q, DC) = W(C_s)$$

    Example 4.11. *Under bag semantics, each derivation corresponds to a distinct answer annotated by a single monomial. This means that the answer $\langle d2, d2 \rangle$ is computed three times leading to three answers, $a_1 = a_2 = a_3 = \langle d2, d2 \rangle$, each of which annotated, respectively, with one of the monomials M_1, M_2 and M_3. For this example, consider CBM and CBS. The inconsistency degrees of these three answers can then be computed as follows: $CBS(a_i, Q, \mathcal{I}, DC) = 3$, for $i \in [1, 3]$, and $CBM(a_1, Q, \mathcal{I}, DC) = 5$, $CBM(a_2, Q, \mathcal{I}, DC) = 6$ and $CBM(a_3, Q, \mathcal{I}, DC) = 4$. We have also $TBM(a_1, Q, \mathcal{I}, DC) = TBS(a_1, Q, \mathcal{I}, DC) = 3$.*

A summary of the definitions of these measures is reported in table 4.2.

Given a query Q and an instance $\mathcal{I}$ and a set of denial constraints DC, $Q(\mathcal{I}^{\Gamma \cup \Upsilon})$ can be computed in polynomial time in data complexity [49, 74]. Hence, all these measures

Bag semantic answers

		TBA		CBA	
Answer	Prov	TBM	TBS	CBM	CBS
$\langle d2, d2\rangle$	M_1	3	3	5	3
$\langle d2, d2\rangle$	M_2	3	3	6	3
$\langle d2, d2\rangle$	M_3	2	2	4	3
$\langle d4, d4\rangle$	1	0	0	0	0

Set semantic answers (TBA)

Answer	Prov	TSM_{max}	TSM_{min}	TSS_{max}	TSS_{min}
$\langle d2, d2\rangle$	$M_1 + M_2 + M_3$	3	2	3	2
$\langle d4, d4\rangle$	1	0	0	0	0

Set semantic answers (CBA)

Answer	Prov	CSM_{max}	CSM_{min}	CSS_{max}	CSS_{min}
$\langle d2, d2\rangle$	$M_1 + M_2 + M_3$	6	4	3	3
$\langle d4, d4\rangle$	0	0	0	0	0

Figure 4.5: Answers of query Q_{ex} over hdb^{Γ} and their inconsistency degrees for each respective inconsistency measure.

of inconsistency degree of answers of a query Q over $\mathcal{I}$ ($Q(\mathcal{I}_{DC})$) can be computed in polynomial in the size of the database instance $\mathcal{I}$.

We consider TBA be the set of measures of inconsistency degrees from tuple-based measures class and CBA be the set of measures from constraint-based measures.

This approach goes beyond the traditional methods that give only information about if a tuple is consistent or inconsistent. So, our approach extends these traditional methods and enables us to compare tuples deeply according to their inconsistencies, each one to the other.

Given a conjunctive query Q, each measure of inconsistency degree of tuple based approach as defined above is bounded by the size of Q (i.e, $|Body(Q)|$). Also, in the case where Q is a self-join free query the set semantics and the bag semantics coincide. The lemma 4.1 shows these two properties.

Lemma 4.1. *Let $\mathcal{I}$, Q and DC be respectively an instance, a conjunctive query and a set of denial constraints, for each tuples $t \in Q(\mathcal{I})$*

1. *$\alpha(t, I, Q, IC) \leq |Body(Q)|$ with $\alpha \in TBA$*
2. *If Q is also self-join free then: $TBM(t, I, Q, IC) = TBS(t, I, Q, IC)$, $TSM_{max}(t, I, Q, IC) = TSS_{max}(t, I, Q, IC)$ and $TSM_{min}(t, I, Q, IC) = TSS_{min}(t, I, Q, IC)$*

Proof. 1. Easy. $Q : Q(X) \leftarrow P_1(X_1), ..., P_n(X_n), \phi$. Each monome M in $Q(\mathcal{I}_{IC})(t)$, $\forall t \in Q(\mathcal{I})$, is in a form $a * \Pi_{i=1}^{n} P_i(v(X_i))$ with v a valuation for Q and $a \in \mathbb{N}$. So, $|Var(M)| \leq |W(M)| \leq n = |Body(Q)|$.

2. $Q : Q(X) \leftarrow P_1(X_1), ..., P_n(X_n), \phi$ a $\mathcal{CQ}$ free self-join $\Rightarrow$ there is no valuation v of Q where $id(v(X_i)) = id(id(X_j))$ with $P_i \neq P_j$, so each monome in $Q(\mathcal{I}_{IC})(t)$

is in form $a \times x_1 \times ... \times x_m$ with $a \in \mathbb{N}$ and $\{x_1, ..., x_n\}$ is the set of variables (identifiers of inconsistent tuples). There is no repetition, thus the bag semantics and set semantics match one with another.

□

While our approach is orthogonal to that of CQA in general (as explained in Section 3), our notion of consistency is much stronger than the notion of consistent answers in CQA as stated by the following lemma (lemma 4.2).

Lemma 4.2. *Let $\mathcal{I}, Q, DC, \alpha$ be an instance, an union of conjunctive query and a set of denial constraints, a measure of inconsistency degrees among measures defined above, respectively.*
$\forall t \in Q(\mathcal{I})$ we have:
$\alpha(t, Q, \mathcal{I}, DC) = \alpha(t, Q, \mathcal{I}, DC) = 0 \Rightarrow t$ is a CQA.

Lemma 4.2 is straightforward to prove as any answer that has inconsistency degree equal to 0 is computed from tuples that do not violate any constraint. As these tuples do not involve any violation, they belong to all the repairs of database regardless of the repair semantics. Clearly, the set of CQA can be larger by including the tuples that involve violations and leveraging a given repair semantics.

Example 4.12. *Consider the following database instance $\mathcal{I}$ with an example of constraint and query:*

$$DC = \{C_1 : \leftarrow R(x,y,z), R(x,y_1,z_1), y \neq y_1\}$$
$$Q(z) : -R(x,y,z)$$

R

A	*B*	*C*	*Prov*	
1	*a*	1	C_1	t_1
2	*b*	2	C_1	t_2
1	*c*	1	C_1	t_3
3	*a*	3	1	t_4
2	*g*	4	C_1	t_5

Consider repair by deletion that is the most repair semantics considered with class of denial constraints [22]. The following set of repairs are obtained:

- $Rep_1 = \{t_1, t_2, t_4\}$
- $Rep_2 = \{t_1, t_5, t_4\}$
- $Rep_3 = \{t_3, t_2, t_4\}$
- $Rep_4 = \{t_3, t_5, t_4\}$

As t_4 does not violate any constraint in DC, then it belongs to all the repair sets. The set of consistent query answers of Q over $\mathcal{I}$ in presence of DC is:

$$CQA(Q, \mathcal{I}, DC) = \{\langle 1\rangle, \langle 3\rangle\}$$

since these two answers can be computed over any repair among $\{Rep_1, Rep_2, Rep_3, Rep_4\}$.

Now, consider the measure CSS_{min} (any measure can be chosen and the same analyses are done). Our approach returns the query answers, as follows: $\{\langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle\}$ with their inconsistency degrees

- $CSM_{min}(\langle 1\rangle, Q, \mathcal{I}, DC) = 1$
- $CSM_{min}(\langle 2\rangle, Q, \mathcal{I}, DC) = 1$
- $CSM_{min}(\langle 3\rangle, Q, \mathcal{I}, DC) = 0$

- $CSM_{min}(\langle 4 \rangle, Q, \mathcal{I}, DC) = 1$

As one can note, in our case only $\langle 3 \rangle$ is consistent. It is also in the set of consistent query answers. But, opposite to the CQA approach, all the query answers are considered and returned, each one with its inconsistency degrees.

4.5 Conclusion

This chapter has presented the measures of inconsistency proposed in this thesis leveraging why-provenance and polynomial semiring provenance. We have categorized these measures into two classes: tuple-based measures and constraint-based measures. These proposed measures of inconsistency are intuitive. They also enable the comparison of tuples, each one to the other, according to their inconsistency degree. Their theoretical complexities are polynomial in data complexity.

The next chapter presents the different top-k algorithms developed to rank query answers and enumerate them in a specific order of their inconsistency.

Chapter 5

INCONSISTENCY-DRIVEN QUERY ANSWERING

In this chapter, we study top-k ranking of inconsistency-aware tuples as defined below. For this purpose, we leverage the set of measures of inconsistency introduced in the Chapter 4. We consider as input an annotated instance $\mathcal{I}^{\Upsilon \cup \Gamma}$, where each base tuple t of a relation R is annotated with the monomial $\mathcal{I}^{\Upsilon \cup \Gamma}(R)(t)$ (c.f. Definition 4.4). Let α be one of the previous measure of inconsistency degrees defined above, the main idea is to use α as a scoring function over the results of a query Q where the score of an output t of Q is given by $\alpha(t, Q, \mathcal{I}, DC)$. The goal is then to rank the answer tuples while taking into account the inconsistency degrees of the base tuples contributing to the answers. The fundamental computation problem is then to be able to efficiently enumerate (part of) query answers in a specific order w.r.t. their inconsistency degrees. We focus on one particular instance of this problem where the goal is to return the query results with the top k scores, hereafter called inconsistency-aware top-k ranking.

Informally, a top-k query Q is a query enabling to compute the first k answers of Q with the highest/lowest scores. These scores are computed using an aggregate scoring function. Depending on the order of the top-k, either k answers with the highest scores or the k answers with lowest scores, we can define separately two types of top-k query: Bottom up top-k(look for the k answers with lowest scores) and Top down top-k queries (look for the k answers with highest scores). Below, we define formally a top-k query in the context of measures of inconsistency degrees. In the rest of paper, we assume that $\iota \in \{\uparrow, \downarrow\}$.

Definition 5.1 (Top-k queries)**.** *Let $\mathcal{I}$, Q and DC be respectively a database instance, a conjunctive query and a set of denial constraints over the same database schema. Let k be an integer, let α be one of measure of inconsistency degrees defined above. The **Bottom up top-k** (resp. **Top down top-k**) query answers of a query Q using the inconsistency measure α, denoted by $Q^{k,\alpha\downarrow}(\mathcal{I})$ (resp. $Q^{k,\alpha\uparrow}(\mathcal{I})$), is defined as follows:*
(i) $Q^{k,\alpha\ \iota}(\mathcal{I}) \subseteq Q(\mathcal{I})$,
(ii) $|Q^{k,\alpha\ \iota}(\mathcal{I})| = Min(k, |Q(\mathcal{I})|)$, *and*
(iii) $\forall (t_1, t_2) \in Q^{k,\alpha\ \iota}(\mathcal{I}) \times (Q(\mathcal{I}) \setminus Q^{k,\alpha\ \iota}(\mathcal{I}))$, *we have:*

- $\alpha(t_1, Q, \mathcal{I}, DC) \leq \alpha(t_2, Q, \mathcal{I}, DC)$ *for Bottom up top-k*
- $\alpha(t_1, Q, \mathcal{I}, DC) \geq \alpha(t_2, Q, \mathcal{I}, DC)$ *for Top down top-k*

Condition *(i)* and *(ii)* ensure that a top-k query $Q^{k,\alpha\ \iota}(\mathcal{I})$ computes at most k answers of $Q(\mathcal{I})$ while condition *(iii)* ensures that the computed answers are the best answers

in $Q(\mathcal{I})$ w.r.t. the inconsistency measure α. The function α is called the scoring function of the top-k problem. We use the notation $Q^{k,\alpha}$ instead of $Q^{k,\alpha,\iota}$, when the orders do not matter.The following example illustrates a top-k query over our running example.

Example 5.1. *Assume $k = 1$ and $\alpha = CBM$. Continuing with the database hdb and the query Q_{ex} as in Figure 4.1, we have: $Q_{ex}^{k,\alpha,\downarrow}(hdb) = \{\langle d4, d4\rangle\}$ and $Q_{ex}^{k,\alpha,\uparrow}(hdb) = \{\langle d2, d2\rangle\}$.*

The complexity of evaluation of a top-k query is mainly depending on the nature of the scoring function [85]. The scoring functions can be divided into two main classes: those functions that are monotonic and those ones that are non-monotonic.

Definition 5.2 (Monotonic function w.r.t. aggregate operator $\circ$)**.** *Let A, B be two sets such that B is totally ordered. Let f be a function from $A \to B$. Let $x, y, z, v \in A$ and let $\circ$ an intern operator in A called the aggregate scoring operator, respectively. The function f is monotonic w.r.t $\circ$, if and only if*

$$f(x) \leq f(y) \wedge f(z) \leq f(v) \Rightarrow f(x \circ z) \leq f(y \circ v)$$

The aggregate scoring operator $\circ$ is used to aggregate scores of tuples used to compute an answer with either join($\bowtie$) or intersection($\cap$) operators. As example of monotonic function with an aggregate operator, we have the identity function from natural numbers set (i.e, $identity : \mathbb{N} \to \mathbb{N}$ such that $identity(x) = x$ with $x \in N$) that is monotonic with addition ($+$) as aggregate score operator; since $identity(x) = x \leq identity(y) = y \wedge identity(z) = z \leq identity(v) = v \Rightarrow identity(x + z) = x + z \leq identity(y + v) = y + v$. Any function that is not monotonic is called a non-monotonic function.

Lemma 5.1. *Let $MS = \{CBM, CSM_{min}, CSM_{max}, TBM, TSM_{min}, TSM_{max}\}$ and $NMS = \{CBS, CSS_{min}, CSS_{max}, TBS, TSS_{min}, TSS_{max}\}$. A scoring function that associates to each tuple t a score computed using a measure of inconsistency degrees from MS (respectively, NMS) is monotonic w.r.t. $+$ (respectively, non monotonic).*

The proof of the above lemma is trivial since the measures in MS can be reduced to the identity function from $\mathbb{N}$ with sum $(+)$ as aggregate operator; however, the measures in NMS are equivalent to cardinal function from subsets of violated constraints or subsets of tuples that involve in violation of constraints ($Card : 2^{\Upsilon} \to \mathbb{N}$ for constraint based approach measures or $Card : 2^{\Gamma} \to \mathbb{N}$ for tuple based approach measures) with the set union as aggregate operator.

In the rest of this chapter, we use the terms score and inconsistency degrees equivalently.

Consider t an answer of a query Q over a database instance $\mathcal{I}$. The set of derivations of t is the minimum (under inclusion operator) subsets from $\mathcal{I}$ from which t can be computed using Q. Denoted by $deriv(t, Q, \mathcal{I})$, the set of derivations of t is formally defined as follows:

$$deriv(t, Q, \mathcal{I}) = \{S \subseteq \mathcal{I} : t \in Q(S) \text{ and } \nexists S' \subset S \text{ and } t \in Q(S')\}$$

It is trivial to see that when the bag semantics query answers is considered, the set of derivations of an answer is always a singleton.

Definition 5.3 (Order Compatible (O.C))**.** *Let $\alpha, \mathcal{I}, Q, DC$ be a measure of inconsistency degrees, an instance, a conjunctive query, and a set of denial constraints, respectively.*

- *The measure α is Order Compatible with Bottom up top-k, denoted $O.C^{\uparrow}$, if only if*

$$\alpha(t, Q, \mathcal{I}, DC) = \min_{S \in deriv(t,Q,\mathcal{I})} \alpha(t, Q, S, DC)$$

- *α is Order Compatible with Bottom up top-k, denoted $O.C^{\downarrow}$, if only if*

$$\alpha(t, Q, \mathcal{I}, DC) = \max_{S \in deriv(t,Q,\mathcal{I})} \alpha(t, Q, S, DC)$$

The Order Compatible property enables to exploit algorithms of top-k designed to measures in bag semantics for the cases of set semantics.

Let's denote by $A = \{CSS_{min}, CSM_{min}, TSS_{min}, TSM_{min}\}$ and $B = \{CSS_{max}, CSM_{max}, TSS_{max}, TSM_$
By definition we have:

1. The measures in A are $O.C^{\uparrow}$
2. The measures in B are $O.C^{\downarrow}$

There exists a naive algorithm enabling to compute the top-k answers of a given query Q with their scores regardless of the class of Q and the scoring function. This naive algorithm computes all the answers of the given Q then sorts them and chooses the k first ones. We shortly note this algorithm NA. The NA algorithm computes a large number of query answers that do not have any utility. It is, then, an efficient top-k algorithm.

The rest of this chapter is organized as follows:

- section 5.1 presents the different dimensions of the problem and the cost model mainly used.
- section 5.2 is dedicated to the top-k algorithms developed for constraint-based measures.
- section 5.3 presents top-k algorithms developed for tuple-based measures.
- section 5.3 concludes this chapter.

5.1 Cost-based query answers enumeration

This section discusses two dimensions of the top-k query answering problem: query answers semantics and the query language. Then, it introduces a cost model, consisting to a new cost function and a new class of top-k algorithms, evaluating the top-k algorithms.

5.1.1 Problem Dimensions

Mainly the problem is to enumerate the query answers in order w.r.t. their inconsistency degrees. For the designing of an algorithm to fix this problem, the following dimensions have to be analyzed: **Query answers semantics** and **Query language**.

Query answers semantics

We investigate two semantics of query answers: bag semantics and set semantics.

In bag semantics of query answers: each query answer is computed in one alternative way with its score(i.e, $\forall t$:answer, $|deriv(t, Q, \mathcal{I})| = |\{S_1\}| = 1$). As, for each query answer t there exists only one possible derivation, then the belonging of t to the top-k answers, once it is computed, is less harder than the case where there exits many possible derivations.

In the case of set semantics of query answers, many alternatives (finite derivations) exist to compute the same query answer arising in many scores for the same answer(i.e, $\forall t$:answer, $|deriv(t, Q, \mathcal{I})| = |\{S_1, \ldots, S_p\}| \geq 1$ with $p \in \mathbb{N}^*$). Each derivation S_i, with $i \in 1, \ldots, p$, gives a possible inconsistency degree of t (that is $IC_i = \alpha(t, Q, S_i, DC)$). So, we need another function f to aggregate these possible inconsistency degrees of t to obtain the inconsistency degree of t (that is $f(IC_1, \ldots, IC_p)$). So, it is harder to decide the belonging of a query answer t, computed from a derivation d, to the top-k answers based only from this derivation d of t.

So, a top-k algorithm for set answers semantic can be used for bag answers semantic since bag semantic is a special case of set semantic (i.e, the case where the set of derivations is a singleton). But, as we show in the rest of this chapter, it is harder to design a top-k algorithm for set semantic than bag semantic. The following example (example 5.2) illustrates this distinction.

Example 5.2. *In this example, we assume that there exist three denial constraints $DC = \{C_1, C_2, C_3\}$. We consider below two relations R_1 and R_2, of an annotated instance $\mathcal{I}$. We consider the conjunctive query $Q(z) : -R_1(x), R_2(x, z)$. The column prov contains the set of violated constraints.*

R_1

	A	*prov*
t_1	0	C_1
t_2	1	C_1C_3
t_3	2	C_1C_2

R_2

	A	B	*prov*
t_4	0	a	1
t_5	1	a	C_2C_3
t_6	2	b	C_2

The goal is to enumerate first the most consistent query answers.

- *Consider now bag semantics query answers and the measure CBS (any measure of bag semantics query answers can be chosen). The query answers is then: $Q(\mathcal{I}) = [\langle a\rangle : C_1, \langle a\rangle : C_1C_2C_3^2, \langle b\rangle] : C_1C_2^2$; we denote by a_i the identifier of the answer at position i in $Q(\mathcal{I})$, so $Q(\mathcal{I}) = [a_1, a_2, a_3]$; we have $Q(\mathcal{I}^Y)(a_1) = C_1$, $Q(\mathcal{I}^Y)(a_2) = C_1C_2C_3^2$ and $Q(\mathcal{I}^Y)(a_3) = C_1C_2^2$. So, we have $CBS(a_1, Q, \mathcal{I}, DC) = 1$, $CBS(a_2, Q, \mathcal{I}, DC) = 3$ and $CBS(a_3, Q, \mathcal{I}, DC) = 2$. Once a_1, a_2, a_3 computed, each one from only one derivation (a_1 from $[t_1, t_4]$, a_2 from $[t_2, t_5]$ and a_3 from $[t_3, t_6]$), the order a_1, a_3, a_2(most consistent answers first) can be established based on CBS measure.*

- *For set semantics, $Q(\mathcal{I}) = \{\langle a\rangle, \langle b\rangle\}$. We have $Q(\mathcal{I}^Y)(\langle a\rangle) = C_1 + C_1C_2C_3^2$ and $Q(\mathcal{I}^Y)(\langle b\rangle) = C_1C_2^2$. Consider the measure CSS_{max}, then $CSS_{max}(\langle a\rangle, Q, \mathcal{I}, DC) = 3$ and $CSS_{max}(\langle a\rangle, Q, \mathcal{I}, DC) = 2$. The top-2 query answers according to CSS_{max} are then $[\langle b\rangle, \langle a\rangle]$. As one can easily note, $\langle a\rangle$ is computed from two derivations $\{d_1 = [t_1, t_3], d_2 = [t_2, t_4]\}$. Depending on the derivation considered the order $[\langle b\rangle, \langle a\rangle]$ can be permitted. Assume that $\langle a\rangle$ is computed first from d_1 and we decide to establish order between $\langle b\rangle$ and $\langle a\rangle$ according to CSS_{max}, then we obtain the following order: $[\langle a\rangle, \langle b\rangle]$ since based only to d_1 the inconsistency degrees of $\langle a\rangle$ with CSS_{max} is equal to 1. This*

leads to an incorrect result. So, for set semantics, it is difficult to rely only on one derivation of query answers to fix order in which query answers are computed according to a measure.

Query language

In the rest of the thesis, we consider the relational algebra language. The expressiveness of conjunctive queries is equivalent to that of relational algebra queries. Depending on the operators of relational algebra used in the query Q, the algorithm used to enumerate query answers in order of their scores can be more complex. An algorithm that enumerates query answers in order of their score can be designed in analyzing the following groups of operators:
Selection (σ), Projection (π), Union ($\cup$): In this case, the algorithm simply consists of accessing tuples in the right order according to their scores.

Example 5.3. *Consider Example 5.2 and the query* $Q_1(z) : -R_2(x, z), x > 0 \equiv \pi(\sigma_{A>0}(R_2))$. *The set of query answers is* $\{\langle a\rangle, \langle b\rangle\}$ *and the inconsistency degrees of* $\langle a\rangle$ *and* $\langle b\rangle$ *are* 2 *and* 1, *respectively.*

Join ($\bowtie$), Intersection ($\cap$): For the case of join and intersection, an answer is computing by joining a set of tuples. Each tuple has its score, so we need an aggregate scoring function $\circ$ to aggregate these different scores. An algorithm that enumerates answers in a specific order of their score, has to consider the nature of this aggregate scoring function. Since the difficulty of designing an algorithm, that enumerates in a specific order the query answers according to their inconsistency degree, depends to the fact that a measure of inconsistency α is monotonic or non-monotonic w.r.t. $\circ$.

5.1.2 Cost models

We introduce a new class of algorithms called SBA and a new cost function. The SBA algorithms defines a set of algorithms that focus mainly and only on the scoring column to evaluate the top-k query. The new cost function enables us to better compare algorithms in SBA than the classical I/O cost function in literature [85]. In fact, the classical I/O cost model is sensitive to the non determinism caused by the read of tuples having the same score. As the class of algorithms considered (SBA) does not have any information about join attributes, the order in which these tuples(having the same score) should not have an improve on the cost of the algorithm.

are read can have influence on the classical I/O cost.

Semi-blind algorithms

In this section, when we mention *prov* attribute so we refer to the attribute that contains the annotation value, i.e, the set of constraints violated by the concerned tuple in monomial form.

We define $\mathcal{S}_\mathcal{A}$ as a database schema where each table R has a set of attributes $\mathcal{A}$. Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two relational instances over $\mathcal{S}_\mathcal{A}$, we define equivalent instances under attributes $\mathcal{A}$: $\mathcal{I}_1 \equiv_\mathcal{A} \mathcal{I}_2$ iff $\forall R \in \mathcal{S}_\mathcal{A}$, $\pi_\mathcal{A}(\mathcal{I}_1(R)) = \pi_\mathcal{A}(\mathcal{I}_2(R))$ with $\pi_\mathcal{A}(R)$ the projection over table R on attributes $\mathcal{A}$.

Let $Q(X) \leftarrow R_1(X_1), \ldots, R_n(X_m)$ and $\mathcal{I}$ be respectively a conjunctive query and an instance over schema $\mathcal{S}_{\{prov\}}$. Let AL be an algorithm that allows to compute the

answers of $Q^{k,\alpha}(\mathcal{I})$, with $\alpha \in \{CBS, CBM\}$. A join test J of AL when processing $Q^{k,\alpha}$ over an instance $\mathcal{I}$ has the following form $J = t_1 \bowtie \ldots \bowtie t_m$ where $t_i \in \mathcal{I}(R_i)$ with $i \in [1, m]$. A score of a join test is defined as follows: $\alpha(J) = \alpha(t_1 \bowtie \ldots \bowtie t_m)$. We define $jPath(AL, Q, \mathcal{I})$ as the sequence of joins test performed by the algorithm AL when evaluating $Q^{k,\alpha}$ over $\mathcal{I}$. The intuition is that $jPath(AL, Q, \mathcal{I})$ enables to capture the *behavior* of the algorithm AL when it evaluates $Q^{k,\alpha}$ over $\mathcal{I}$. Formally, $jPath(AL, Q, \mathcal{I}) = [J_1, ..., J_n]$, where $J_i = t_1^i \bowtie \ldots \bowtie t_m^i$, with $t_j^i \in \mathcal{I}(R_j)$ and $j \in [1, m]$. We define $label(J_i) \stackrel{\text{def}}{=} (\pi_{prov}(t_1^i), \ldots, \pi_{prov}(t_m^i))$. We are now ready to give the formal definition of the class of *Semi-Blind Algorithms* (SBA), the algorithms that use only information from the *prov* column when processing inconsistency-aware queries.

Definition 5.4 (SBA algorithms)**.** *Let $Q^{k,\alpha}$ be a top-k query, let $\mathcal{I}_1, \mathcal{I}_2$ be two instances, and let AL be an algorithm such that: $jPath(AL, Q, \mathcal{I}_2) = [J'_1, ..., J'_{n_2}]$ and $jPath(AL, Q, \mathcal{I}_1) = [J_1, ..., J_{n_1}]$. The algorithm AL belongs to the class SBA iff:*
$\mathcal{I}_1 \equiv_{\{prov\}} \mathcal{I}_2 \Rightarrow label(J_i) = label(J'_i)$, *for* $i \in [1, max(n_1, n_2)]$

According to this definition, an algorithm $AL \in SBA$ will have a similar behavior when evaluating a query $Q^{k,\alpha}$ over different instances that are equivalent under *prov*, i.e., AL will explore the same sequence of join tests but may stop earlier or later depending on the success or failure of the join tests. The outcome of this latter test is related to the content of the join attributes of each specific input instance and remains independent from the *prov* column. As one can easily notice, TopINC belongs to SBA. Indeed, it only relies on the information given by the *prov* column without exploiting any auxiliary information.

Region-based cost function

A natural metric to measure the performance of an algorithm AL is to compute the number of tuples of the input relations accessed by AL, denoted $cost(Al, Q, \mathcal{I})$.

Let $J = t_1 \bowtie \ldots \bowtie t_m$, we denote by $tuple(J) = \{t_1, \ldots, t_m\}$ the set of tuples involved in a join test J. Consider an algorithm AL with $jPath(AL, Q, \mathcal{I}_1) = [J_1, ..., J_{n_1}]$. Then, the cost of AL is given by: $cost(Al, Q, \mathcal{I}) = |\bigcup_{i=1}^{n_1} tuple(J_i)|$. The following theorem states that there exists no algorithm in the class SBA that is optimal w.r.t. the *cost* metric defined above.

Theorem 5.1. *Let $Q^{k,\alpha}$ be a top-k query. Then, we have: $\forall Al_1 \in SBA, c\exists Al_2 \in SBA$ and an instance $\mathcal{I}$ such that, $cost(Al_1, \mathcal{I}, Q) > cost(Al_2, \mathcal{I}, Q)$.*

Proof. W.l.o.g, assume that $m = 2$ (i.e., Q is a join between two relations R_1 and R_2). Take two integers $l > 1, p > 1$ such that $k \leq l * p$. We build an instance $\mathcal{I}_1$ as follows:

- The relation $\mathcal{I}_1(R_1)$ contains two subsets of tuples (the l-tuples and the p-tuples): l tuples $t_1^l, \ldots, t_l^l$ with $\pi_{prov}(t_i^l) = \pi_{prov}(t_j^l), \forall i, j \in [1, l]$ and p tuples $t_1^p, \ldots, t_p^p$ with $\pi_{prov}(t_i^p) = \pi_{prov}(t_j^p), \forall i, j \in [1, p]$. We take $\pi_{prov}(t_1^p) \neq \pi_{prov}(t_1^l)$.
- The relation $\mathcal{I}_1(R_2)$ contains two subsets of tuples (the l-tuples and the p-tuples): l tuples $t_1'^l, \ldots, t_l'^l$ with $\pi_{prov}(t_1^p) = \pi_{prov}(t_i'^l) \forall i \in [1, l]$ and p tuples $t_1'^p, \ldots, t_p'^p$ with $\pi_{prov}(t_1^l) = \pi_{prov}(t_i'^p), \forall i \in [1, p]$.

The *prov* values are choosed such that they satisfy the following condition: $\alpha(\pi_{prov}(t_1^l) \times \pi_{prov}(t_1^p)) < min(\alpha(\pi_{prov}(t_1^l) \times \pi_{prov}(t_1^l), \alpha(\pi_{prov}(t_1^p) \times \pi_{prov}(t_1^p))$. Consider an algorithm $Al_1 \in SBA$ with $jPath(Al_1, Q, \mathcal{I}) = [J_1, ..., J_{n_1}]$. We exhibit the following cases regarding the first test join J_1:

- Case when $J_1 = t_1^p \bowtie t_2^p$ or $J_1 = t_1^p \bowtie t_2^l$ (i.e., Al_1 starts reading a p-tuple from R_1). We construct an instance $\mathcal{I}_2$ such that $\mathcal{I}_2 \equiv_{prov} \mathcal{I}_1$ and all the answers of Q over $\mathcal{I}_2$ are exactly those answers obtained by joining l-tuples of $\mathcal{I}_2(R_1)$ with p-tuples from $\mathcal{I}_2(R_2)$, i.e., any other join test between tuples of $\mathcal{I}_2(R_1)$ and tuples of $\mathcal{I}_2(R_2)$ will evaluate to false. Clearly, Al_1 is not optimal to evaluate $Q^{k,\alpha}$ over $\mathcal{I}_2$ because Al_1 starts to process join test that does not lead to any output answer as a result of the fact that $label(jPath(Al_1, Q, \mathcal{I}_2)[1]) = label(J_1)$ (since $\mathcal{I}_2 \equiv_{prov} \mathcal{I}_1$) and hence Al_1 reads at least one useless tuple (the p-tuple t_1^p which, by contruction of $\mathcal{I}_2$, do not contribute to any answer). It is easy to see that a round robbin algorithm that alternate reading l-tuples from R_1 and p-tuples from R_2 and performing join tests between the tuples loaded in the memory is optimal. Indeed, to maximize the generated answers for a given cost s (i.e., reading s base tuples), the best strategie is to read $\lceil s/2 \rceil$ base l-tuples from R_1 and $\lfloor s/2 \rfloor$ base p-tuples from R_2 (or inversely)[1], which enables to compute the maximal number of $\lceil s/2 \rceil \times \lfloor s/2 \rfloor$ answers. The behavior of such an algorithm Al_2 is given by $jPath(Al_2, Q, \mathcal{I}) = [J'_1, \ldots, J'_k]$ with $J'_1 = {t_1}^l \bowtie {t'_1}^p$, $J'_2 = {t_2}^l \bowtie {t'_1}^p$, $J'_3 = {t_1}^l \bowtie {t'_2}^p$, $J'_4 = {t_2}^l \bowtie {t'_2}^p$, $J'_5 = {t_3}^l \bowtie {t'_1}^p$, $J'_6 = {t_3}^l \bowtie {t'_2}^p$ ...

  Since, Al_2 reads the minimal number of base tuples to compute k answers from $\mathcal{I}_2$ and Al_1 reads at least one useless base tuple, we can conclude that $cost(Al_1, \mathcal{I}_2, Q) > cost(Al_2, \mathcal{I}_2, Q)$.

- Case when $J_1 = t_1^l \bowtie t_2^l$ or $J_1 = t_1^l \bowtie t_2^p$ (i.e., Al_1 starts reading an l tuple from R_1). This case is the dual of the first one and can be proved following the same reasoning while inverting the roles of p-tuples and l-tuples when building $\mathcal{I}_2$ and $Al2$.

□

The intuition behind the above theorem 5.1 is that the SBA algorithms need to make a non-deterministic choice among the join tests that have equivalent score. We illustrate this case by relying on a corner case of an instance $\hat{\mathcal{I}}$ containing exclusively consistent tuples. Consider a query Q over n inputs $R_1, \ldots R_n$ such that $|Q(\hat{\mathcal{I}})| = 1$. Consider now the evaluation of the query $Q^{1,CBS}$ by SBA algorithms. Since all the join tests among the tuples of the input relations will have the same score, an SBA algorithm needs to make a non-deterministic choice among the elements of $R_1 \times \ldots \times R_n$ to decide in which order the join tests will be performed. Hence, the *best* algorithm Al would luckily pick the right tuples in the first round of choice which leads to an optimal cost: $cost(Al, \hat{\mathcal{I}}, Q) = n$. The worst-case algorithm Al' might end up with the least good cost after exploring the entire cartesian product of the inputs, which leads to $cost(Al, \hat{\mathcal{I}}, Q) = \sum_{i=1}^{n} |R_i|$ (i.e., the algorithms Al' needs to read the entire inputs). Consequently, we argue that it is not worthwhile to distinguish between SBA algorithms w.r.t. to the order of exploring join tests with equivalent score (since this is a non-deterministic choice). We formalize this notion using *regions*, defined as maximal

[1]This is because the optimum of the function $f(x) = sx - x^2$, for a constant s, is given by $x = s/2$.

	O.SBA	E-k-answers	Complexity
TopINC	✓	✓	$O(n^m)$
TupIncRank	✓	✓	$O(n^m)$
TopIncMem	NaN	✓	$O(n^\delta + k)$
TopINCDE	✓	✗	$O(n^m)$
TopMultiSet	NaN	✓	$O(n^\delta + log(n) * k)$
TopIncSet	NaN	✓	$O(n^\delta * k)$
NA	✗	✗	$O(n^m + m * log(n))$

Table 5.1: Algorithms with their characteristics; n is the size of the database instance, m is the size of the query and k is the number of answers to compute; $\delta <= n$ measures the number of intermediary answers (it is 1 if the query is acyclic)

sub-sequences of join tests with equivalent score, and we define a new metric based on the number of regions explored by the algorithm. The region-based cost enables us to get ride of lucky choices when comparing the performances of SBA algorithms.

Below, we define the notion of a region. Let $jPath(AL, Q, \mathcal{I}) = [J_1, ..., J_n]$. A region of $jPath(AL, Q, \mathcal{I})$ is a maximal subsequence of $jPath(AL, Q, \mathcal{I})$ made of join tests with equal inconsistency degree. More formally, a sequence $[J_l, ..., J_p]$, with $l \leq p$, is a region of $jPath(AL, Q, \mathcal{I}) = [J_1, ..., J_n]$ iff $l, p \in [1, n]$, and: *(i)* $\alpha(J_i) = \alpha(J_j), \forall i, j \in [l, p]$, and *(ii)* $\alpha(J_{l-1}) \neq \alpha(J_l)$ and $\alpha(J_{p+1}) \neq \alpha(J_p)$. We define $Regs(Al, Q, \mathcal{I})$ to be the set of regions of $jPath(AL, Q, \mathcal{I})$. We define the cost model $cost^\nabla(Al, Q, \mathcal{I})$ as the number of regions explored by the algorithm Al during processing of query Q over $\mathcal{I}$:

$$cost^\nabla(Al, Q, \mathcal{I}) = |Regs(Al, Q, \mathcal{I})|$$

We call this cost model REGION-BASED COST FUNCTION. The introduced cost model $cost^\nabla(Al, Q, \mathcal{I})$ conveniently prevents an algorithm to compute an answer that can be dropped after to the top-k answers, thus avoiding more useless I/O operations.

In the following sections, we present the different top-k algorithms. A summary of these algorithms is depicted in table 5.1. The columns **O.SBA**, **E-k-answers**, **Complexity** mean that the algorithm is optimal in SBA with region-based cost function; the algorithm computes exactly k answers without computation of any additional answer; and the data complexity of the algorithm, respectively.

5.2 Algorithms for constraint-based measures

This section presents top-k algorithms for constraint-based measures. These algorithms can be divided into two classes: Memory-based algorithms and Disk-based algorithms. These algorithms and measures for which they are designed are depicted in figure 5.1.

5.2.1 Disk-based algorithms

An algorithm is in the class of Disk-based if it only optimizes the input/output operations on disk.

TopINC algorithm

In this section, we present our top-k ranking algorithm, called TopINC, for top-k queries under both inconsistency measures CBM and CBS. A general idea behind existing

Query Answer semantics	Bag semantics		
Query Language	$\sigma, \pi, \cup$	$\bowtie, \cap$	
Agg.Score.Fun. properties	NaN	Monotone	Non-Monotone
Measures	*All*	*CBM*	*CBS*
Algorithms	TopINC, FA, TA, NRA[51]	TopINC, J^* [125], rankJoin [84], Take2 [140]	TopINC, TopIncMem

Query Answer semantics	Set semantics					
Query Language	$\sigma, \pi, \cup$		$\bowtie, \cap$			
Agg.Score.Fun. properties	$O.C \uparrow$	$O.C \downarrow$	Monotone		Non-Monotone	
			$O.C \uparrow$	$O.C \downarrow$	$O.C \uparrow$	$O.C \downarrow$
Measures	CSS_{min}, CSM_{min}	CSS_{max}, CSM_{max}	CSM_{min}	CSM_{max}	CSS_{min}	CSS_{max}
Algorithms	TopMultiSet, TopINCDE		TopMultiSet, TopINCDE		TopIncSet, TopINCDE	

Figure 5.1: Algorithms for constraint-based measures

rank join algorithms [85] is to process the tuples of the relations involved in a given query in a specific order by considering at each step the most promising tuples at first. However, when the scoring function is not monotonic with respect to the join operator, which is the case for CBS due to single-occurrences, it is not straightforward to identify the order in which tuples should be processed.

The core intuition of our top-k ranking algorithm consists of using an index based on the inconsistency measures to access the most promising tuples at each step of query processing. More precisely, we build for each relation R, an associated index, denoted $ind(R)$, whose nodes are labeled by a subset of the constraints. More precisely, each node B of $Ind(R)$ is labeled with $label(B) \subseteq DC$. A node B stores the set of tuples's ids of R, denoted $B(R)$, that violate **exactly** the set of constraints $label(B)$, i.e., the set $B(R) = \{t \in R \mid VC(\mathcal{I}, DC, t) = B\}$. We call B a bucket of the index $Ind(R)$ and the set $B(R)$ the bucket content.

Example 5.4. *In our example, the buckets of the index are labeled with subsets of the constraints $DC = \{C_1, C_2, C_3\}$. For instance, a bucket B_0, with $label(B_0) = \emptyset$, will store the consistent tuples $B_0(D) = \{t_1\}, B_0(S) = \{t_5, t_6\}$ and $B_0(V) = \{t_8\}$ of the relations D, S and V. The labels of the buckets and the content of the buckets are given below.*

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
$l(B)$	$\emptyset$	$\{C_1\}$	$\{C_2\}$	$\{C_3\}$	$\{C_1, C_2\}$	$\{C_1, C_3\}$	$\{C_2, C_3\}$	$\{C_1, C_2, C_3\}$
$B(D)$	$\{t_1\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{t_2\}$	$\emptyset$	$\emptyset$	$\emptyset$
$B(S)$	$\{t_5, t_6\}$	$\{t_3\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{t_4\}$	$\emptyset$	$\emptyset$
$B(V)$	$\{t_8\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{t_7\}$	$\emptyset$

Each index $ind(R)$ is defined as an ordered list of the nodes containing fragments of R, where nodes are ordered with respect to the cardinality set of their labels, i.e., $B \leq B'$ iff $|label(B)| \leq |label(B')|$.

Example 5.5. *The following indexes are associated with the relations D, S and V of our example database: $Ind(D) = [B_0, B_4]$, $Ind(V) = [B_0, B_6]$ and $Ind(S) = [B_0, B_1, B_5]$.*

We now describe the Algorithm in detail. Let Q be a query $Q(u) \leftarrow R_1(u_1), \ldots, R_m(u_m)$. In order process the query $Q^{k,\alpha}$, with $\alpha \in \{CBS, CBM\}$, the algoritm TopINC (c.f., Algorithm 1) takes as input: the query $Q^{k,\alpha}$, the indexes $ind(R_i)$, with $i \in [1, m]$, one for each input relation and the set $ViolC$ of the violated constraints obtained by unioning the labels of all the buckets in the indexes. The algoritm uses as many temporary

buffers $HR_i, i \in [1, m]$ as the number of indexes. Each temporary buffer contains the bucket contents. In addition, the algorithm uses a vector jB of size m storing the ids of the buckets that need to be joined during the course of the algorithm. The algorithm TopINC follows a level-wise sequencing of the iterations, where each level denotes the inconsistency degree of the answers computed at this level (c.f., lines 4-8 of algorithm 1). Level 0 means that consistent tuples are processed while the subsequent level l indicates the number of constrainst that are considered for the violated tuples. When processing a given level l, the algorithm resets the variable $curVSet$, used to keep track of the violated constraints while exploring the input relations at level l, and makes a recursive call to the IterateJoin procedure (line 7). For each level l, the IterateJoin procedure (Algorithm 2) explores the input relations sequentially from R_1 to R_m (lines 11 to 14). For each input relation R_p, IterateJoin uses the index $ind(R_p)$ to identify the buckets of R_p that are worthwhile to consider for the join (i.e., the buckets to be loaded in $jB[p]$), i.e., those buckets whose label size's is less than l (line 5). The relevant bucket ids of input relations are loaded in the jB buffer (line 13 of algorithm 2) and when R_p is the last input relation (i.e., $p = m$) (line 15) a join is performed between the buffers of jB (lines 17-20 of the algorithm 2) in order to compute the answers with inconsistency degree equal to the current level l. Note that the variable $curVSet$ will contain duplicate occurrences if $\alpha = CBM$ (line 10) and single occurrences if $\alpha = CBS$ (line 7). It enables us to keep track of the current level of inconsistency while exploring the inputs. When intermediate inputs are explored, the IterateJoin algorithm ensures that $|curVSet|$ does not exceed the current inconsistency level l (line 5 and line 11). When the last input is processed, a join is performed between the buckets in jB only if $|curVSet| = l$ (line 15) which ensures that the computed answers have l as inconsistency degree.

Algorithm 1: TopINC

```
   Input  : ViolC: set of violated constraints
            Q^{k,α} : a top-k query over R_1, ..., R_m with α ∈ {CBM, CBS}
            ind(R_1), ..., ind(R_m) : indexes of the input relations
   Output: Res: k best answers w.r.t. α
   Data structures: HR_1, ..., HR_m: input buffers;
   jB : an array of size m
1  begin
2      Res=[] be an empty list ;
3      /* Contr.  violated by current answer                          */
4      level:=0 ;
5      while l ≤ |ViolC| ∧ Res.size < k do
6          curVSet := ∅ ;
7          IterateJoin(1, level, curVSet) ;
8          level:= level + 1 ;
9      return Res
```

Example 5.6. *We assume that the input relations are processed in this order: D, S and V. We illustrate the processing of query $Q_{ex}^{2,CBS}(\mathcal{I})$ by TopINC. Figure 5.2 exemplifies the iterations of the algorithm. The gray cells, in each step, denote the newly read tuples in that step. The final result (Level 3) is reported at the bottom of the Figure 5.2. Starting from level 0, the selected buckets are $jB = [B_0, B_0, B_0]$, thus leading to join the contents of these buckets only at the very beginning. The contents of $HV(B_0)$, $HS(B_0)$ and $HD(B_0)$ are shown in Figure 5.2. The first answer corresponding to the join of the above buffers is found (i.e, $res = [\langle d4, d4\rangle]$). The TopINC continues to read in selected buckets in jB. It then tries to read in B_0 of V but no additional tuples, then it moves to read in B_0 of V. Next, the tuple $\langle 02, d2, 5\rangle$ is loaded in $HV(B_0)$ but no additional answer is found. As there is no additional answers (because k=2)*

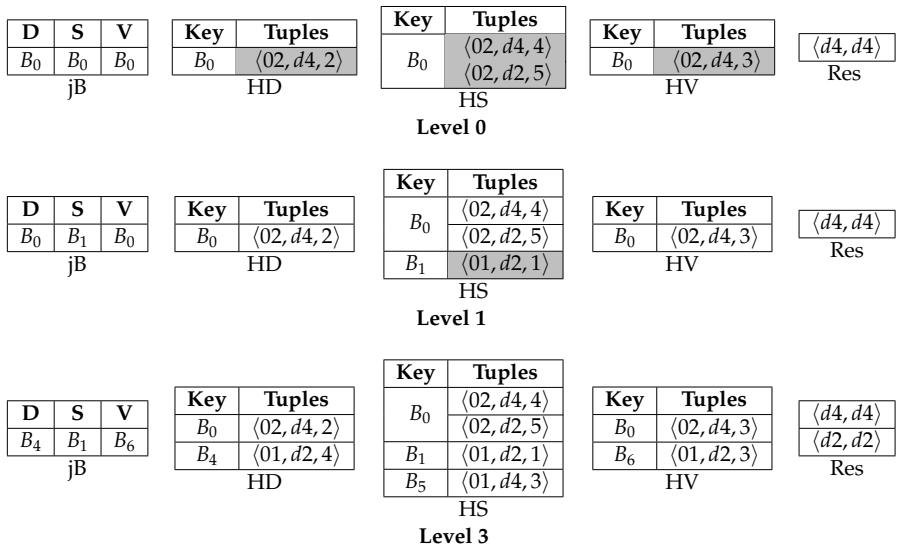


Figure 5.2: Illustraive example of TopINC

and all tuples are read in selected buckets, TopINC jumps to the next level, i.e, the level 1*. At this level, the first selected buckets are* $jB = [B_0, B_1, B0]$*, thus leading to read the only one tuple of bucket* B_1 *of relation V into* $HV(B_1)$*. But no additional answer is found and no others buckets in level* 1 *can be selected bringing TopINC to level* 2*. Finally, TopINC processes level* 2 *and then level* 3 *as shown in Figure 5.2 and halts when it reaches* $|res| = 2$*.*

The following two lemmas state the correctness and the worst case complexity of TopINC, respectively.

Lemma 5.2. *Let* $\alpha \in \{CBM, CBS\}$*. The algorithm TopINC computes correctly* $Q^{k,\alpha}(\mathcal{I})$*.*

Proof. The TopINC algorihtm proceeds one level at a time. It starts from inconsistency degree 0 to upper inconsistency degrees. At each inconsistency degree d fixed, it looks for all the joins of buckets (from join relations) where the union of their labels has cardinality d; the join is only performed among the tuples within the buckets found. It stops processing when k answers are found otherwise TopINC continues to look for the remaining joins of buckets for the same inconsistency degree d; if there exists no other join of buckets that leads to inconsistency degree d, the TopINC moves forward to inconsistent degree $d + 1$. Hence, this processing ensures that the K answers output by TopINC are correct. □

Lemma 5.3. *Let* $Q(X) \leftarrow R_1(X_1), \ldots, R_m(X_m)$ *be a conjunctive query, let* $s = |X|$ *and let* k *be an integer. A query* $Q^{k,\alpha}$ *is evaluated by the TopINC in time* $O(n^m)$ *and in space* $O(|\mathcal{I}| + k \times s)$*.*

Proof. This lemma 5.3 is a consequence of the level-wise approach followed by TopINC which ensures that query answers are computed in the ascending order of their inconsistency degree. Hence, TopINC strictly computes the answers that belong to the output (and the algorithm stops when k answers are computed, hence the spatial complexity in $O(|\mathcal{I}| + k * s)$ with the maximal size of an answer). The time complexity $O(n^m)$ is due of the fact that tuples are read one by one according to their scores; with every reading of one tuple, a join test is done to search a new answer. So, one can easily denote that the join algorithm can quickly become a nested loop algorithm: so the worst complexity is $O(n^m)$ with $n = |\mathcal{I}|$ and m the size of the input conjunctive query Q. □

Algorithm 2: IterateJoin

Input: p, level, curVSet

```
1  begin
2      /* Index of input relation p                                   */
3      Let idx := ind(R_p);
4      i:=1 ;
5      while i ≤ idx.size ∧ |label(idx[i])| ≤ level do
6          if α = CBS then
7              curVSet := curVSet ∪ label(idx[i])
8          else
9              /* ⩒ stands for bag union                            */
10             curVSet := curVSet ⩒ label(idx[i])
11         /* Case input R_p is not the last                         */
12         if p < m ∧ |curVSet| <= level then
13             jB[p]:=idx[i];
14             IterateJoin(p + 1, level, curVSet);;
15         if p = m ∧ |curVSet| = level then
16             jB[p]:=idx[i];
17             /* Compute the join from jB                           */
18             ans := HR_1(jB[1]) ⋈ ... ⋈ HR_m(jB[m]) ;
19             /* Add the results to Res up to k                     */
20             Res.add(ans) ;
21             if Res.size = k then
22                 EXIT;
23         i := i + 1;
```

Unsurprisingly, and like most of state of the art top-k join algorithms that do not use specific knowledge related to the join attributes (e.g., see [85, 131]), the worst case time complexity of TopINC is in $O(|\mathcal{I}|^m)$, with m the number of input relations in Q. Interestingly, the previous lemma also provides a tighter upper bound regarding the space complexity. This lemma is a consequence of the level-wise approach followed by TopINC which ensures that query answers are computed in the ascending order of their inconsistency degree. Hence, TopINC strictly computes the answers that belong to the output (and the algorithm stops when k answers are computed).

The following theorem shows the optimality of TopINC in the SBA algorithms.

Theorem 5.2. *For any instance $\mathcal{I}$ and any top-k conjunctive query $Q^{k,\alpha}$, we have:*

$$cost^{\nabla}(\textsf{TopINC}, Q, \mathcal{I}) \leq cost^{\nabla}(AL, Q, \mathcal{I}), \forall AL \in SBA$$

Proof. Let $Q^{k,\alpha}$, with $\alpha \in \{CBM, CBS\}$, be a top-k query over an instance $\mathcal{I}$. Assume that there exists an algorithm $AL \in SBA$ that outputs $Q^{k,\alpha}(\mathcal{I})$ with $cost^{\nabla}(AL, Q, \mathcal{I}) < cost^{\nabla}(\textsf{TopINC}, Q, \mathcal{I})$. The intuition behind our proof is that if TopINC explores a region Z that is not explored by AL then AL is not correct (i.e., it do not correctly compute the top-k answers). Note that, when TopINC starts performing join tests with inconsistency degree d, it fully explores the region (i.e., performs all the possible join tests with inconsistency degree d) before moving to another region. In addition, TopINC processes

the regions sequencially in increasing order of their inconsistency degrees (i.e, starting from region of inconsistency degree 0 to upper degrees of inconsistency). Let Z be the region having the biggest inconsistency degree, noted d_z, among the regions explored by AL. Two cases occur: *(i)* either AL has exhaustively explored all the regions with inconsistency degree $< d_z$ and in this case $cost^{\nabla}(AL, Q, \mathcal{I}) \geq cost^{\nabla}(\mathsf{TopINC}, Q, \mathcal{I})$ (because TopINC will also explore the same regions or a subset of them), or *(ii)* AL skips some regions with inconsistency degree $< d_z$. In this case, one can prove that AL is incorrect. Indeed, since $AL \in SBA$, one can build an instance $\mathcal{I}' \equiv_{prov} \mathcal{I}$ such that AL outputs incorrect answers when it evaluates $Q^{k,\alpha}$ over $\mathcal{I}'$. □

TopINCDE algorithm

We propose TopINCDE algorithms to evaluate top-k queries w.r.t set semantics query answers while optimizing the input/output operations on disk. The algorithm TopINCDE is designed to work with measures $CSS_{min}, CSS_{max}, CSM_{min}, CSM_{max}$.

The main idea of TopINCDE is to use any top-k algorithm(called the kernel algorithm) designed in a context of bag semantics answers; then, we use this algorithm to enumerate query answers in order of their inconsistency degrees and eliminate the duplicate answers on the fly. The kernel algorithm can be TopINC algorithm or any top-k algorithm in bag semantics.

But, one needs additional conditions with measures to enable TopINCDE to compute correctly the top-k answers for these measures. This condition is about order compatibility between measure and the top-k order. In fact, if a measure is not order compatible with the top-k order needed, TopINCDE gives an incorrect set of answers. The following example(example 5.7) illustrates this scenario.

Example 5.7. *Consider the following instance $\mathcal{I}$ already annotated with the set of denial constraints $DC = \{C_1, C_2, C_3\}$:*

R

	A	E	
r_1	a	1	$C1$
r_2	a	2	$C1$

S

	B	E	
s_1	b	1	1
s_2	b	2	$C2C3$
s_3	c	1	$C2$

$Q(x, y) : -R(x, z), S(y, z)$
Consider the measure $\alpha = CSM_{max}$ and the measure $\alpha_1 = CSM_{min}$. Compute the top-2 the most consistent (i.e, the bottom up top-2) answer of query Q considering α_1 first.

Q

	A	B	
t_1	a	b	C_1
t_2	a	b	$C_1C_2C_3$
t_3	a	c	C_1C_2

$\alpha(\langle a, b\rangle, Q, \mathcal{I}, DC) = 3$
$\alpha(\langle a, c\rangle, Q, \mathcal{I}, DC) = 2$
$\alpha_1(\langle a, b\rangle, Q, \mathcal{I}, DC) = 1$
$\alpha_1(\langle a, c\rangle, Q, \mathcal{I}, DC) = 2$

The algorithm TopINCDE computes the top-2 answers in this order $[t_1 = \langle a, b\rangle : C_1, t_3 = \langle a, c\rangle : C_1C_2]$. For the measure α_1, the result is correct since α_1 is $O.C^{\uparrow}$.

But, for α the correct order of answers is $[t_3 = \langle a, c\rangle : C_1C_2, t_2 = \langle a, b\rangle : C_1C_2C_3]$. Since α is $O.C^{\uparrow}$ then TopINCDE can not compute correctly the top-2 answers consistent answers with α.

Lemma 5.4. *Let α be a measure of inconsistency degrees, so*

- *if α is $O.C^{\uparrow}$ then TopINCDE computes correctly $Q^{k,\alpha\uparrow}(\mathcal{I})$*
- *if α is $O.C^{\downarrow}$ then TopINCDE computes correctly $Q^{k,\alpha\downarrow}(\mathcal{I})$*

Proof. Trivial. The algorithm TopINCDE takes as kernel the algorithm TopINC with eliminating of duplicate on the fly. As α is order compatible with the top-k category, when an answer, that is not yet computed, is computed then the current score is its true score and one can keep it in the top-k answers. So, when the order compatibility of a scoring function α is checked TopINCDE computes correctly the set of the top-k answers if and only if TopINC runs correctly. We shown in 5.2 that TopINC runs correctly, so TopINCDE correctly computes the k answers when the property of order compatibility is checked. □

The following theorem shows that TopINCDE visits fewer regions than any other algorithm in SBA.

Lemma 5.5. *Let $Q, \mathcal{I}$ be a conjunctive query and an instance, respectively. For any algorithm $Al \in SBA$, $cost^{\nabla}(Al, Q, \mathcal{I}) \geq cost^{\nabla}(\textsf{TopINCDE}, Q, \mathcal{I})$*

Proof. As TopINCDE is based on TopINC, it iterates the regions one by one based on their scores and when the k answers are found the running of the algorithm is stopped. So any other algorithm in SBA that computes the top-k answers (in set semantic of query answers) goes through at least the number of regions visited by TopINCDE $-$trivial$-$. Then, the optimality of TopINCDE in SBA with $cost^{\nabla}$ as cost measure. □

The main deficiency of this new algorithm (TopINCDE) is the fact that additional answers (the duplicate answers) can be computed and thrown out later.

The TopINCDE has the same time theoretical complexity than TopINC.

5.2.2 Memory-based algorithms

The class of Memory-based algorithms focuses on optimizing operations in main memory. Most of existing Memory-based top-k algorithms are based on monotonic function [85]. Also, these algorithms consider only full conjunctive query. We propose, in this section, Memory-based top-k algorithms that are either based on non monotonic function or extend full conjunctive query with projection. We assume that any query in this section is an equi-conjunctive query. The algorithms in this class take advantage from efficient techniques developed to evaluate efficiently the join operation.

TopIncMem algorithm

The TopIncMem algorithm is a top-k algorithm developed to work with CBS measure. We apply the same principle as in the case of TopINC but we assume that the database is loaded in main memory. The algorithm is depicted in algorithm 3.

The algorithm TopIncMem takes as input a tree (denoted by T) corresponding to the join tree of the input conjunctive query and the input database. We associate with each node of the tree, the underling index of the relations used to build this node. As in the case of TopINC, it is denoted by *buckets*. The *buckets* represents the set of violated set of constraints by each node of the tree.

During the building of the tree, on each node is associated a map structure with keys from the set of join attribute values between the node and its parent. This map builds a local index which is used to accelerate the join.

Algorithm 3: TopIncMem

Input : T: tree from Q and I, k: integer, buckets: list of set of violated constraints

Output: k most consistent answers of $Q(I)$ w.r.t CBS

```
1  /* initialization of global variables                    */
2  level := 0;
3  while Res.size < k do
4      combinations := bucketComb(level, buckets);
5      for comb ∈ combinations do
6          T' := filter(T, comb);
7          tempRes := compute(T', k − Res.size);
8          Res.addAll(tempRes);
9          if Res.size = k then
10             break;
11     level := level + 1;
12 return topK ;
```

The algorithm TopIncMem is levelwise algorithm. Each level corresponding to the value of inconsistency degrees with CBS. It start by the level 0. Given a level l, TopIncMem looks for the combinations of violated sets for each node such that the cardinality of their union corresponds to this l (done by the function *bucketComb* in line 4). After that, for each combination taken, the tree is filtered according to this combination (shown in line 6) and after that the Yannakakis's algorithm can be used to enumerate query answers in this filtered tree. Only query answers that violate these constraints are computed (in line 7).

Example 5.8. *Consider the instance($\mathcal{I} = \{R_1, R_2, R_3, R_4\}$) and the conjunctive query Q in figure 5.3.a with the set of denial constraints $DC = \{C_1, C_2, C_3\}$. We directly annotate each tuple with the set of constraints it violates. The illustration of TopIncMem running is depicted in figure 5.4. Vertically, we have the iteration on levels and horizontally we have the iteration of possible combinations for the same level. At the first level (level=0) only one combination exists that corresponds to $(1, 1, 1, 1)$ and this combination leads to the first answer in Res. After that, TopIncMem moves to the level 1, as shown in 5.4, the first two combinations of level 1 leads to no additional answers. The algorithm TopIncMem proceeds in this way until the k first answers are obtained.*

Lemma 5.6. *The algorithm TopIncMem computes correctly the top-k answers that are the most consistent/inconsistent w.r.t. the measure of inconsistency CBS.*

Proof. By absurd. Assume that there is an answer a_1 in the set answers computed using TopIncMem such that there exists an answer a_2 that is not in the set of answers returned by TopIncMemand the degrees of inconsistency of a_2 is less than that of a_1, i.e, $CBS(a_1, Q, \mathcal{I}, DC) = l_1 > CBS(a_2, Q, \mathcal{I}, DC) = l_2$. The TopIncMem algorithm explores all the combinations of violated constraints, such that the cardinality of the union of each combination gives the current level and the answers corresponding to this level are computed by filtering the join tree. As the level start from 0 to the number of constraints, it means that when an answer (as a_2) is not yet computed any other already computed answer by TopIncMem(as a_1) has an inconsistency degrees less than that of

R_1

X	Y	Z	
0	1	a	1
0	1	b	C_1

R_2

X	z_1	
0	5	1
2	6	C_3
0	1	C_2C_3
1	3	C_1C_2

R_3

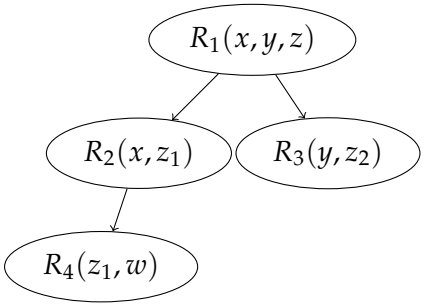
Y	z_2	
1	b	1
4	c	C_1C_3
3	d	C_2

R_4

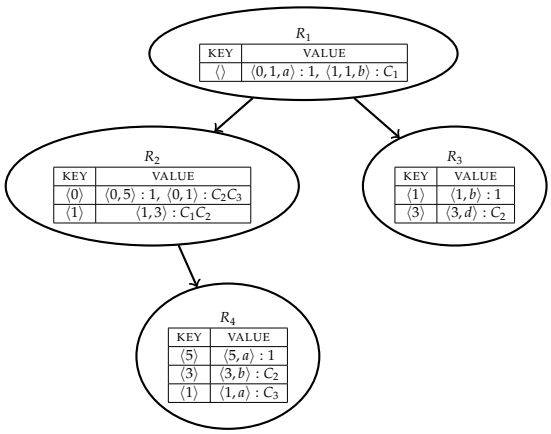
z_1	W	
5	a	1
3	b	C_2
1	a	C_3

$$Q(x, y, z, z_1, z_2, w) : -R_1(x, y, z), R_2(x, z_1), R_3(y, z_2), R_4(z_1, w)$$

a.



b.



c.

Figure 5.3: Example of Data to illustrate top-k Memory-based algorithms (a.). Tree example (b.). Tree with data (c.)

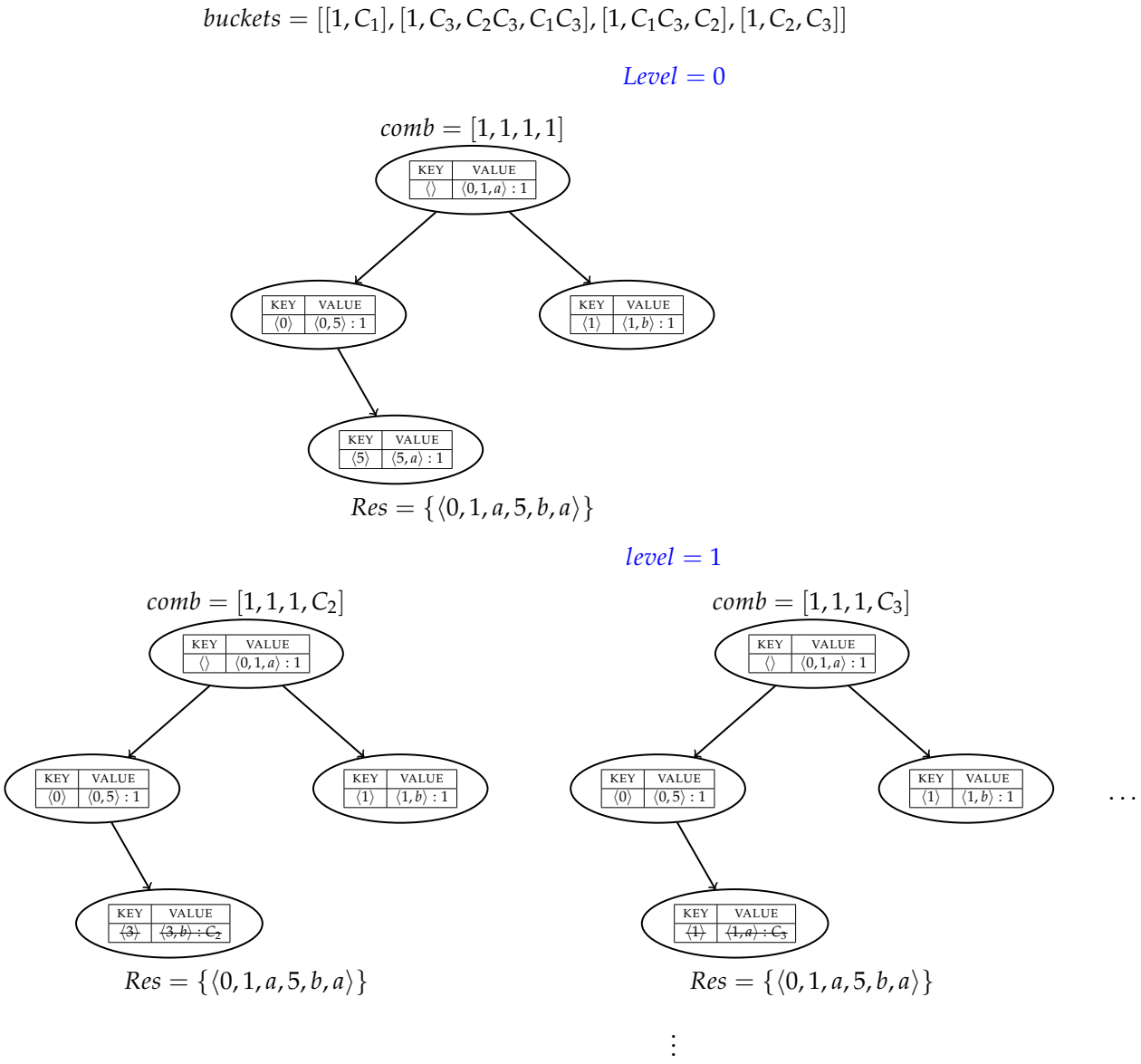


Figure 5.4: TopIncMem illustration

the first one, i.e, $l_1 \leq l_2$. Absurd. So, TopIncMem computes correctly the first answers that are the most consistent with respect to the measure of inconsistency CBS. □

Lemma 5.7. *The algorithm TopIncMem runs in a complexity $O(2^{|Q|*|DC|} \times |\mathcal{I}|^{\delta} + k)$ (so, $O(|\mathcal{I}|^{\delta} + k)$ in data complexity) where Q is the input query, I the input database instance, DC the input set of denial constraints and δ the tree fractional width of Q, respectively. The spacial complexity is $O(2 * |\mathcal{I}|^{\delta} + k)$.*

Proof. For each R_i relation in join, the index size (size of $ind(R_i)$) is bounded by the size of the set of subsets formed from DC, so the size is $2^{|DC|}$. To find the combinations of a given level, we have to search in the space $ind(R_1) \times \ldots \times ind(R_m)$ with $m = |Q|$, so if all the combinations are visited then we have visited $2^{|DC|} * \ldots * 2^{|DC|} = 2^{|Q|*|DC|}$ combinations. For each combination, the algorithm filters the join tree and computes the answers corresponding to the current level, this operation can be done in $|\mathcal{I}|^{\delta}$ corresponding to the largest intermediary result. So the theoretical complexity is bounded by $2^{|Q|*|DC|} * |\mathcal{I}|^{\delta} + k$ with k the set of number of answers kept (so, the data complexity is $|\mathcal{I}|^{\delta} + k$). □

This algorithm can be effective in using for CBS than TopINC if a large main memory is available and the database can be entirely and quickly loaded in this main memory.

TopMultiSet algorithm

The algorithm TopMultiSet is designed to compute the k answers of given query in $\mathcal{I}$ that are the most consistent (or inconsistent) with respect to the measures of inconsistency degrees CSM_{min} (or CSM_{max}). It is based on the algorithm developed in [48]. The algorithm designed in [48] assumes a monotone scoring function, and enumerates answers in order of their scores. It is designed in a case of a bag semantics of query answers. The is composed to two parts: a preprocessing part (performed by the function called *preprocess*) that associate some data structures on each node during tree building from the hypergraph that represents the conjunctive query, and an enumeration part (performed by the function called *enum*) that allows to enumerate answers in order of their scores.

Each node no of the tree represents an input relation or a join result of subset of input relations (intermediary result), denoted $rel(no)$. We denote by $parent(no)$ the node parent of no. Let $comVarP(no)$ the common variables between the node no and its parent. The node no contains a subset of variables from the input query denoted $var(no)$. It contains a map structure, denoted $map(no)$, containing the input data of the relation that this node represents. The keys values of the $map(no)$ is the set of tuples obtaining by making projection on $comVarP(no)$ and in $rel(no)$. Given a key ke, the value corresponding in the $map(no)$ is a priority queue containing all the tuples in $rel(no)$ that have projection value, on $comVarP(no)$ and in $rel(no)$, equal to ke.

The tuples in the priority queue are ordered according to a global score (i.e, score obtained by aggregating score of tuples in join) of tuples. Let $child(no)$ be the child nodes of the node no. Consider t a tuple, we denote par $t.score$ the score of t and the global score of t is denoted by $t.global$. The global score of a tuple t in a left node is equal to the score of the tuple(i.e, when t is a leaf node then $t.score = t.global$). For any tuple t in a internal node $n1$, we have

$$t.global = t.score + \sum_{n0 \in child(n1)} map(n0)[t[comVarP(n0)]].top.global$$

with $t[V]$ the projection value of set of variables in V; and top calls on the priority queue gives the tuple with the larger/smaller global score.

After the preprocessing, the first answer can be already enumerated. Once an answer is enumerated, the tree is updated to prepare the next answer. The update of the tree is done as follows: from the leafs to the root of the tree, for each node $n0$ let t be the tuple used in the current answer; t is in a priority queue Qe; let $[n_1, \ldots, n_p]$ the child nodes of $n0$, then t is joined with $[map(comVarP(n_1)).top, \ldots, map(comVarP(n_p)).top]$; t is replaced by p other copies of t such that the i^{th} is joined with

$$[map(comVarP(n_1)).top, \ldots, map(comVarP(n_{i-1})).top, next(map(comVarP(n_i)).top),$$

$$map(comVarP(n_{i+1})).top, \ldots, map(comVarP(n_p)).top]$$

where $next(map(comVarP(n_i)).top)$ is the top element in the priority queue after deleting of the top element of the priority queue. If the $next$ top element of $map(comVarP(n_i)$ is null, i.e, it does not exist a new for the top element then the i^{th} copy of t is ignored. After this update of the tree, a new answer can be computed, if the root node is not empty.

The TopMultiSet algorithm needs the satisfaction of the order compatibility between the top-k order and the scoring function. It performs such that during the enumeration of an answer, its duplicate instances are directly removed, i.e, an answer is computed once. The pseudo code of this algorithm is given in algorithm 4.

Algorithm 4: TopMultiSet

Input : Q: conjunctive query, $\mathcal{I}$: annotated database instance, k: integer, $Attr$: set of attributes projected in Q

Output: Res: k most consistent answers of $Q(I)$ (or inconsistent) w.r.t CSM_{min}, TSM_{min} (or CSM_{max}, TSM_{max})

```
1  T := preprocess(Q, I);
2  Queue := [T];
3  /* a priority queue, elements in Queue are ordered by their first
      element                                                          */
4  while Res.size < k do
5      T' := Queue.pop();
6      /* The best tree, i.e, tree with the best first element, is
          removed and returned                                         */
7      Ans := enum(T');
8      /* The next best element in T' is returned                     */
9      Res.add(Ans);
10     nextTrees := partition(T', Ans, Attr);
11     for T'' ∈ nextTrees do
12         Queue.push(T'');
13 return Res ;
```

First, a preprocessing is done to generate the join tree with all the necessary data structures by the preprocessing algorithm designed in [48], as mentioned by the line 1 of algorithm TopMultiSet. The algorithm enables, in an iterative way, to enumerate one by one the answer in order of their score developed in [48] is used to return at each step the next best answer. As the enumeration algorithm developed in [48] is designed

Algorithm 5: *partition*$(T', Ans, Attr)$

```
1  nextTrees := {};
2  /* let linNodes be an array of nodes of T', i.e, the T' in linear
       form                                                          */
3  prevNodes := [];
4  for i := 1 to linNodes.size do
5      Node := linNodes[i];
6      Let T'' the tree from T' in which ∀j ∈ [1, i − 1], linNodes[j] is replaced by
         prevNodes[j] and Node is replaced by Node in which all the tuples with
         attributes Attributes(Node) ∩ Attr value equal to Ans are removed;
7      Let newNode the node from Node where only tuples with attributes
         Attributes(Node) ∩ Attr value equal to Ans are keeped ;
8      prevNodes.add(newNode);
9      Remove all the isolated nodes in T'';
10     if T'' has the same nodes number than T' then
11         Res.add(T'');
12 return Res ;
```

for a full conjunctive query (or in a context of bag semantics of query answers), at each step we separate the problem into m(with m the size of Q) other join trees and these join trees are sorted according to their first best element in a priority queue as the line 10 of algorithm TopMultiSet shows. Then, in the next round, the best tree (the tree that has the answer of Q with the best score) is popped (in line 6) from *Queue* and its first element is returned as shown in line 8 of algorithm TopMultiSet.

Using dynamic programming techniques [140], the function *Partition* allows to separate the problem into at most m other problems. This separation enables to avoid computation of duplicate answers. Consider *VarPro* the set of variable in projection; let T be the array of nodes of the tree and let $m_1 = |T|$; let $dropDup(t, var, no)$ be a function that removes all the tuples, which have projection value on variable var equal to t, from the node no; consider $keepLast(t, var, no)$ a function that keeps only the tuples, which have projection value on variable var equal to t, from the node no. Assume that the current answer computed is t. The function *Partition* works as follows: convert T into m_1 other arrays of nodes(that are other trees); the i^{th} tree, denoted T_i, is:

$$T_i = \begin{cases} T_i[1] & = & keepLast(t, VarPro, T[1]) \\ & & \vdots \\ T_i[i-1] & = & keepLast(t, VarPro, T[i-1]) \\ T_i[i] & = & dropDup(t, VarPro, T[i]) \\ & & \vdots \\ T_i[i+1] & = & T[i+1] \\ & & \vdots \\ T_i[m_1] & = & T[m_1] \end{cases}$$

In the following example, we illustrate the algorithm TopMultiSet.

Example 5.9. *Consider the following instance.*

- $R_1 = \{\langle a, 1\rangle : 1,\ \langle a, 2\rangle : 1,\ \langle a, 3\rangle : 2\}$

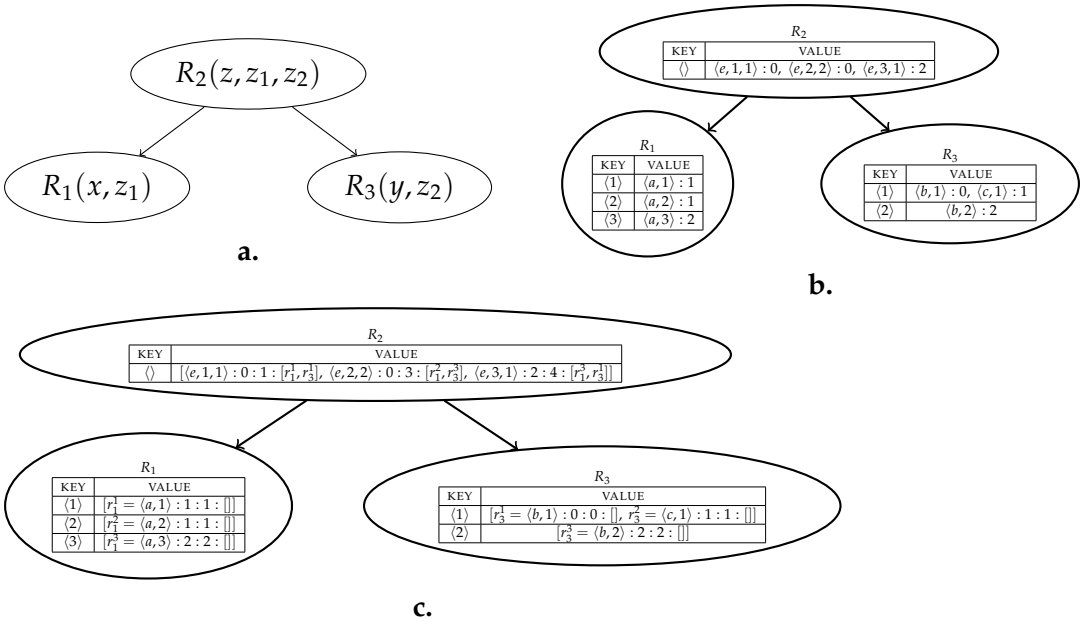


Figure 5.5: Join tree with data for TopMultiSet algorithm and the preprocessing step

- $R_2 = \{\langle e,1,1\rangle : 0,\ \langle e,2,2\rangle : 0,\ \langle e,3,1\rangle : 2\}$
- $R_3 = \{\langle b,1\rangle : 0,\ \langle b,2\rangle : 2,\ \langle c,1\rangle : 1\}$

The value after the «:» gives the score of the tuple. The score means the number of constraints violated by the tuple. Now consider the following conjunctive query

$$Q(x,y,z) : -R_1(x,z_1),\ R_2(z,z_1,z_2),\ R_3(y,z_2)$$

The goal of this example is to find the top-2 answers that is the most consistent answer w.r.t. the measure CSM_{min}. As CSM_{min} is compatible with this order of top-k, we can compute correctly the top-2 using TopMultiSet algorithm.

The join tree of Q is depicted in the figure 5.5.(a), the tree with data of Q is depicted in the figure 5.5.(a). After that, the preprocessing step runs and the first answer is directly computed as shown in 5.5.(c). In the tree 5.5.(b), after the first «:» we have the score (i.e, number of constraints violated) of the tuple, and after the second «:» we have the total best score aggregated from the children nodes (global score). The array after the last «:» contains the addresses of tuples joined with the current tuple. It is interesting to keep that since a tuple t used in a child node is removed from its priority queue Qe. But the tuple t keeps a link with the next best element of Qe.

The illustration of the algorithm TopMultiSet is done in figure 5.6. The first answer is then $\langle a,b,e\rangle : 1$. As we need to compute the two first consistent answers, the algorithm TopMultiSet continues to run. The join tree 5.6.(a) shows the state of the join tree 5.5.(c) after computation of the first answer $\langle a,b,e : 1\rangle$. Once the answer $\langle a,b,e : 1\rangle$ is computed, the current tree 5.6.(a) on which this answer is compute is divided into three other trees (tree 5.6.(b1), tree 5.6.(b2) and tree 5.6.(b3)). In the tree 5.6.(b1), the tuples with projection value on the first column equal to $\langle a\rangle$ are removed; in 5.6.(b1) and 5.6.(b3) are removed tuples with projection value on the first column equal to $\langle e\rangle$ and tuples with projection value on the first column equal to $\langle b\rangle$, respectively. Among these new trees only the tree 5.6.(b3) can give new answers, it is depicted in 5.6.(c) after dangling tuples eliminations. Then the next answer is $\langle a,c,e\rangle : 2$.

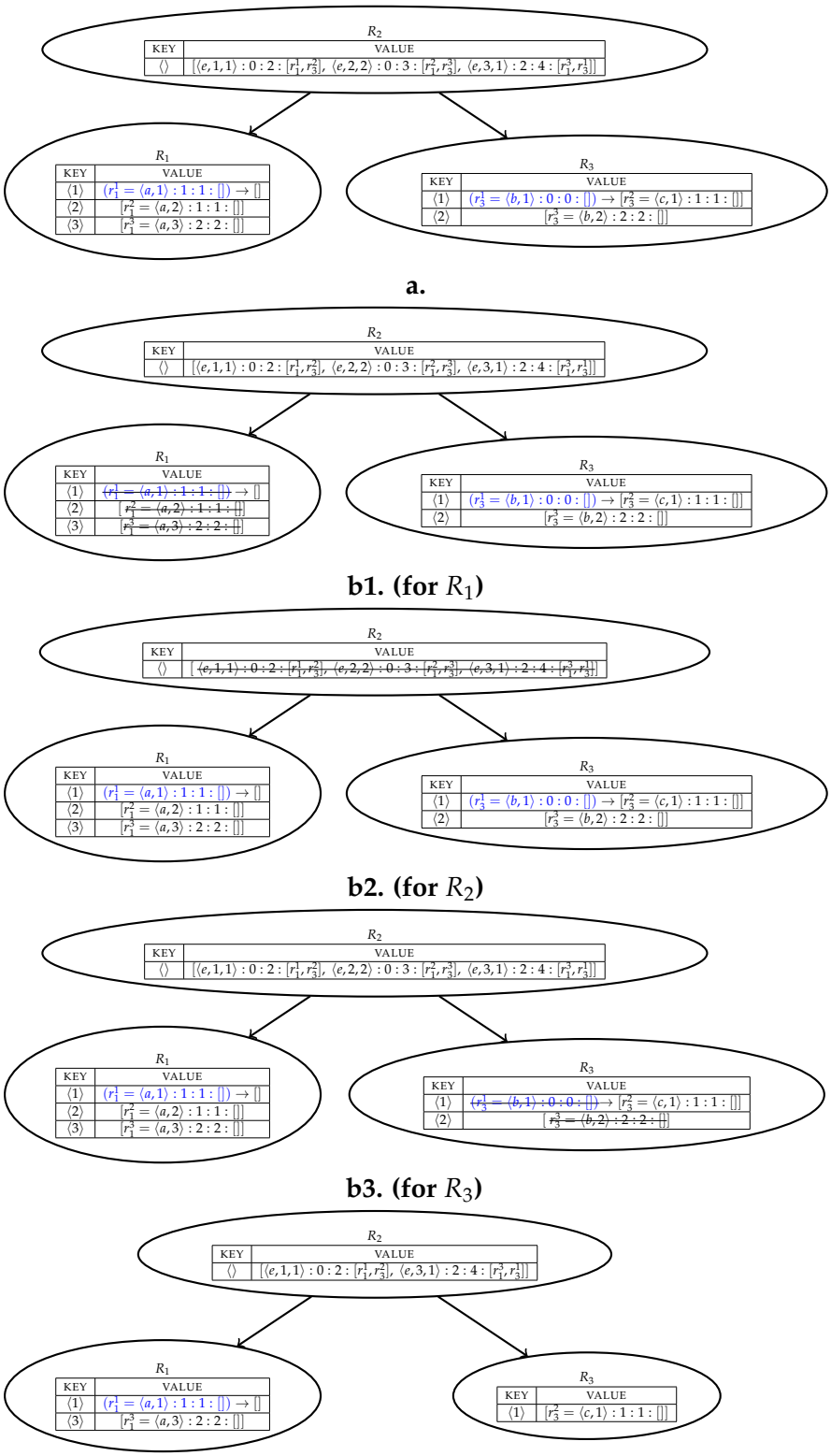


Figure 5.6: Illustration of TopMultiSet algorithm. Partitioning of the tree after computing of the first answer

Lemma 5.8. *The algorithm TopMultiSet computes correctly the k most consistent (resp. inconsistent) answers w.r.t the measures CSM_{min}, TSM_{min} (resp. CSM_{max}, TSM_{max})*

Proof. Assume that there are two a_1 and a_2 ($a_1 \neq a_2$) such that the degree of inconsistency of a_1 by CSM_{min} or by TSM_{min} is strictly greater than that of a_2 and a_1 is in the set of top-k consistent answers computed by TopMultiSet w.r.t the measures $\{CSM_{min}, TSM_{min}\}$ but a_2 is not in this set. That means that:

1. a_1 is computed from a join tree different from the join tree of a_2
2. a_1 is computed from the same join tree than a_2

If (1), it is absurd since the join trees are sorted in a priority queue according to their first element. If (2) also, it is absurd because in the tree join the first element is returned using the same technique as in [48]. Then, the algorithm computes exactly the correct answers. Also by the partition strategy, there is no redundant answer in the answers returned by the TopMultiSet algorithm. □

Lemma 5.9. *The data complexity of TopMultiSet running is $O((|\mathcal{I}|^{\delta} + log(|\mathcal{I}|)) * k)$*

Proof. The maximal intermediary result generating during the join tree building is $|\mathcal{I}|^{\delta}$ and $log(|\mathcal{I}|)$ is for the internal priority queue in the tree join to find the first element of the join tree. For each answer computed, at most $|\mathcal{I}|^{\delta} + log(|\mathcal{I}|)$ operations are done. So, the complexity is bounded by $(|\mathcal{I}|^{\delta} + log(|\mathcal{I}|)) * k$ □

TopIncSet algorithm

Considering, the measures CSS_{min} and CSS_{max}, we obtain function that are not monotonic opposed to $CSM_{min}, TSM_{min}, CSM_{max}, TSM_{max}$ that are monotonic. So, we design a new algorithm based on TopIncMem algorithm to handle the problem of top-k answers w.r.t. CSS_{min} (for the most consistent answers) and w.r.t. CSS_{max} (for the most inconsistent answers).

We proceed as in the case of TopMultiSet algorithm that takes as kernel the algorithm of enumerating developed in [48]. The TopIncSet algorithm takes as kernel the algorithm TopIncMem. When an answer is computed, in the same way as in TopIncMem, the algorithm of partition is used to partition the current join tree (i.e, the tree used to generate the answer) into m other problems (trees). The Different trees are kept in a set data structure, contrary to the case of TopMultiSet where they are kept in a priority queue. The choice of a set rather than a priority queue is the fact that we can not know the best answer in the join tree easily. The algorithm TopIncSet runs as follows: for each combination of violated sets found (as done in algorithm TopIncMem), all the join trees generated by the partitioning operation are visited in an iterative way. Once a tree is chosen, the current combination is used to compute the next answers of the input query.

As one can easily note, this algorithm enable correctly to compute the k first answers that are the most consistent w.r.t. the measure CSM_{min} and the k most inconsistent answers w.r.t. to CSS_{max}.

Lemma 5.10. *The algorithm TopIncSet runs $O(|\mathcal{I}|^{\delta} * k)$ in data complexity.*

Query Answer semantics	Bag semantics		
Query Language	$\sigma, \pi, \cup$	$\bowtie, \cap$	
Agg.Score.Fun. properties	NaN	Monotone	Non-Monotone
Measures	*All*	*TBM*	*TBS*
Algorithms	TupIncRank, FA, TA, NRA[51]	TupIncRank, J^* [125], rankJoin [84], Take2 [140]	TupIncRank(for FSJCQ), NA

Query Ans.seman.	Set semantics					
Query Lang.	$\sigma, \pi, \cup$		$\bowtie, \cap$			
Agg.Score.Fun.prop.	$O.C \uparrow$	$O.C \downarrow$	Monotone		Non-Monotone	
			$O.C \uparrow$	$O.C \downarrow$	$O.C \uparrow$	$O.C \downarrow$
Measures	TSS_{min}, TSM_{min}	TSS_{max}, TSM_{max}	TSM_{min}	TSM_{max}	TSS_{min}	TSS_{max}
Algorithms	TopMultiSet, TopINCDE, NA				TopMultiSet(FSJCQ), NA	

Figure 5.7: Algorithms for tuple-based measures

5.3 Algorithms for tuple-based measures

This section presents top-k algorithms for tuple-based measures. They can be also categorized into two classes: Memory-based algorithms and Disk-based algorithms. These algorithms and measures for which they are designed are depicted in figure 5.7.

5.3.1 TupIncRank algorithm

The TupIncRank algorithm computes the top-k answers that are the most or less inconsistent with respect to the measure TBM. The algorithm TupIncRankcan be used with TBS in the case where the input query is a free self join conjunctive query.

As input, TupIncRank algorithm takes an annotated instance database, a conjunctive query and an integer k(that is the number of the first inconsistent/consistent answers of the query). The database is annotated, in a way that each inconsistent tuple is annotated by 1 and each consistent tuple is annotated by 0.

The algorithm TupIncRank works simply as follows. It maintains a set noted by *incSet* (as shown in line 7), this set contains all the identifiers of relations that have to be inconsistent in the next join performing. Any relation out of this set has its part entirely consistent considered in the next joins to do. The set *incSet* starts with empty set, in other word, in the next join only consistent parts of relations, in join, have to considered. The algorithm TupIncRank iterates, in a descending order according to the number of inconsistent relations considered, the lattice formed by the subsets of the set of relation identifiers in join. For example to have all the answers of a query Q that have inconsistency degrees equal to 2, we have to do join with 2 relations with their parts entirely inconsistent, so the cardinality of *incSet* has to be two. Once the value of *incSet* is found, TupIncRank read one by one the tuples from relations parts considered (as in line 9 and line 17). At each step of read of new tuple the join is performed with the set of tuples already read (at line 13 and at line 24). In line 9, *readNext* function is used. ReadNext function takes a relation R_i and a binary value (0 or 1: $IconsParts[i]$ value) and read the next tuple in R_i with inconsistency 1 or 0 according to $IconsParts[i]$ value. This new tuple is added into the buffer $HR_i[IconsParts[i]]$. Once, TupIncRank found the k answers, its stops query processing and outputs the k answers. Let's illustrate TupIncRank algorithm in the example 5.10.

Algorithm 6: TupIncRank

Input : I: binary annotated instance, k: integer,
$Q(X) : -R_1(X_1), \ldots, R_m(X_m), \phi(X_1, \ldots, X_m)$

Output: k most consistent answers of $Q(I)$ w.r.t TBM (or TBS, Q free self-join CQ)

```
1  /* initialization of global variables                    */
2  HR_i := [];
3  topK := [];
4  remainds := [{1,...,m}];
5  incSets := [∅];
6  while |topK| < k do
7      incSet := incSets.remove(0);
8      performJoin(incSet);
9      if topK.size < k then
10         break;
11     remaind := remainds.remove(0);
12     tempRemoveRemaind := ∅;
13     for e ∈ remaind do
14         tempIncSet := IncSet ∪ {e};
15         tempRemoveRemaind := tempRemoveRemaind ∪ {e};
16         incSets.add(tempIncSet);
17         remainds.add(remaind / tempRemoveRemaind);
18 return topK ;
```

Algorithm 7: performJoin

```
1  inconParts := [];
2  rels := [];
3  for i := 1 to m do
4      if i ∈ incSet then
5          InconsParts[i] := 1;
6      else
7          inconsParts[i] := 0;
8      if HR_i[inconsParts[i]] = nil then
9          tempTup := ReadNext(i, inconsParts[i]);
10         if tempTup ≠ nil then
11             HR_i[inconsParts[i]] := {tempTup};
12             rels.add(i);
13 Add in topK k − topK.size answers from
     Q(HR_1[inconsParts[1]], ..., HR_m[inconsParts[m]]) ;
14 ///* If rels is not empty, otherwise rel is equal to −1          */
15 rel := 1;
16 while rels.IsNotEmpty And |topK| < k do
17     tup := ReadNext(rels[rel], InconsParts[rels[rel]]);
18     if tup = nil then
19         remove rels[rel];
20         rel := ((rel + 1)%(rels.size + 1)) + 1;
21         continue;
22     HR_rel[inconsParts[rels[rel]]] := HR_rel[inconsParts[rels[rel]]] ∪ {tup};
23     Let S_i := HR_i[inconsParts[i]];
24     Add in topK k − topK.size answers from
         Q({S_1, ..., S_rel−1, {tup}, S_rel+1, ..., HR_m}) ;
25     rel := ((rel + 1)%(rels.size + 1)) + 1;
```

Example 5.10. *Continuing with our illustrative database used in the previous examples. The goal of this illustration is to find the 2 first answers that are the most consistent w.r.t the measure TBM and TBS. By R[1] we mean the set of tuples read in R. The annotated database obtained is the following:*

PID	RefD	Date		
02	d4	2	0	t_1
01	d2	4	1	t_2

Diagnosis(D)

PID	RefD	Date		
01	d2	1	1	t_3
01	d2	3	1	t_4
02	d4	4	0	t_5
01	d2	5	0	t_6

Surgery(S)

PID	RefD	Date		
01	d2	3	1	t_7
02	d4	3	0	t_8

Vaccination(V)

With the query Q_{ex}

- $IncSet = \emptyset \quad TBS \equiv TBM \equiv 0$

HD	
Key	Value
0	t_1

HS	
Key	Value
0	t_5

HV	
Key	Value
0	t_8

$$Q(\{D[0], S[0], V[0]\}) \rightarrow TopK = [\langle 04, 04\rangle]$$

HD	
Key	Value
0	t_1

HS	
Key	Value
0	t_5, t_6

HV	
Key	Value
0	t_8

$$Q(\{D[0], \{t_6\}, V[0]\}) \rightarrow TopK = [\langle 04, 04\rangle]$$

- $IncSet = \{D\} \quad TBS \equiv TBM \equiv 1$

HD	
Key	Value
0	t_1
1	t_2

HS	
Key	Value
0	t_5, t_6

HV	
Key	Value
0	t_8

$$Q(\{D[1], S[0], V[0]\}) \rightarrow TopK = [\langle 04, 04\rangle]$$

- $IncSet = \{S\} \quad TBS \equiv TBM \equiv 1$

HD	
Key	Value
0	t_1
1	t_2

HS	
Key	Value
0	t_5, t_6
1	t_3

HV	
Key	Value
0	t_8

$$Q(\{D[0], S[1], V[0]\}) \rightarrow TopK = [\langle 04, 04\rangle]$$

HD	
Key	Value
0	t_1
1	t_2

HS	
Key	Value
0	t_5, t_6
1	t_3, t_4

HV	
Key	Value
0	t_8

$$Q(\{D[0], \{t_4\}, V[0]\}) \rightarrow TopK = [\langle 04, 04\rangle]$$

- $IncSet = \{V\} \quad TBS \equiv TBM \equiv 1$

HD	
Key	Value
0	t_1
1	t_2

HS	
Key	Value
0	t_5, t_6
1	t_3, t_4

HV	
Key	Value
0	t_8
1	t_7

$$Q(\{D[0], S[0], V[1]\}) \rightarrow TopK = [\langle 04, 04\rangle]$$

- $IncSet = \{D, S\} \quad TBS \equiv TBM \equiv 2$

HD

Key	Value
0	t_1
1	t_2

HS

Key	Value
0	t_5, t_6
1	t_3, t_4

HV

Key	Value
0	t_8
1	t_7

$$Q(\{D[1], S[1], V[0]\}) \rightarrow TopK = [\langle 04, 04\rangle]$$

- $IncSet = \{D, V\}$ $TBS \equiv TBM \equiv 2$

HD

Key	Value
0	t_1
1	t_2

HS

Key	Value
0	t_5, t_6
1	t_3, t_4

HV

Key	Value
0	t_8
1	t_7

$$Q(\{D[1], S[0], V[1]\}) \rightarrow TopK = [\langle 04, 04\rangle,\ \langle 02, 02\rangle]$$

The running of TupIncRank algorithm is stopped since two answers are found.

Theorem 5.3. *The algorithm TupIncRank computes correctly the top-k answers that are the most consistent w.r.t measures TBM and TBS.*

Proof. We proof this theorem by absurd . Let $Q(X) : - R_1(X_1), \ldots, R_n(X_n), \phi(X_1, \ldots, X_n)$ be a query. Consider the two answers t_1 and t_2 of Q with inconsistency degrees with measures TBM(also with TBS) equal to a_1 and t_2, respectively, such that $a_1 > a_2$. So there are $S_1 \subseteq \{1, \ldots, n\}$ and $S_2 \subseteq \{1, \ldots, n\}$ such that $|S_1| = a_1$ and $|S_2| = a_2$. We have also, $t_1 \in Q(\{R_i[1] : i \in S_1\} \cup \{R_i[0] : i \in \{1, \ldots, n\}/S_1\})$ and $t_2 \in Q(\{R_i[1] : i \in S_2\} \cup \{R_i[0] : i \in \{1, \ldots, n\}/S_2\})$. Assume that TupIncRank computes t_1 before t_2. If t_1 is computed before t_2 by TupIncRank then S_1 is explored by TupIncRank before S_2. That is absurd since the lattice formed by the set of relations in join is iterated in order of their size. □

Theorem 5.4. *Consider the problem of computing of the k first answers that are the most consistent w.r.t measures TBM and TBS(case where query is fSJ CQ), then*

1. *the algorithm TupIncRank is optimal in the class of SBA with the most measure cost$^{\Delta}$*
2. *no additional answer is computed outside of the k first answers that are the most consistent w.r.t TBM and TBS*

Proof. Let's consider $Q(X) := R_1(X_1), \ldots, R_n(X_n), \Phi(X_1, ..., X_n)$ as the input query. The algorithm TupIncRank is level-wise algorithm. It iterates the lattice formed by the set of name/identifiers of relations in join (i.e, the set $\{R_1, \ldots, R_n\}^2$). Assume that there is an algorithm $Al \in SBA$ that computes the k first consistent answers of Q for measures TBM and TBS with p_1 visited regions and TupIncRank computes the same set of answers of Q with p_2 visited regions such that $p_1 < p_2$. When an algorithm in SBA start to visit a region, all the regions are crossed before starting to visit an other region and no index on join attributes is available. As a region matches a set of join tests with the same score (inconsistency degrees), in the case of TBS and TBM a region corresponds to the cardinality of number of relation totally inconsistent in join. The algorithm TupIncRank iterates the lattice of relations in join from empty set to the set of all relation (i.e, in ascending order). Each set of relations SR selected means that relation in SR have to be completely inconsistent and those are out of this set are completely consistent. So, if Al crosses less regions than TupIncRank after acclaiming the top-k answers then Al skipped and ignored some regions − implies that Al does not correctly the top-k answers. So, either any algorithm of SBA than TupIncRank is incorrect either it crosses more regions or equal than TupIncRank after acclaiming the top-k results. □

The TopINCDE and TopMultiSet algorithms can be used to process top-k algorithm with TSM_{min}, TSM_{max} measures and TSS_{min}, TSS_{max} measures if the query is free self join.

A top-k algorithms can be used to fix some filtering problems such as: compute the first answers that have the inconsistency/inconsistency degrees greater than or less than or equal a given value.

5.4 Conclusion

We designed a set of algorithms to perform top-k query with our proposed measures as scoring functions. We explain the different dimensions to consider for designing a top-k algorithm for these measures. We categorize these algorithms according to their cost models. We shown the optimality of some of these algorithms while given their theoretical complexities. In the next chapter, we present our approach to fix inconsistency in the case where the set of constraints are inconsistent.

Chapter 6

PRESENCE OF INCONSISTENCY IN SET OF CONSTRAINTS

In the previous chapters, we assume that the set of constraints considered is consistent. We present in this chapter preliminary results considering a context in which the set of denial constraints are inconsistent. In particular, we define in this context a set of measures quantifying inconsistency of query answers.

Let DC be a set of denial constraints defined from schema $\mathcal{S}$.

The set DC is inconsistent if only if: $\nexists I \neq \emptyset$, an instance database over the schema $\mathcal{S}$, such that $I \models DC$.

Example 6.1. *Consider a schema $\mathcal{S} = \{R(A, B, C)\}$ and the following set of denial constraints on $\mathcal{S}$:*

- $C_1 : \leftarrow R(x, y, z), x \leq y$
- $C_2 : \leftarrow R(x, y, z), z \geq y$
- $C_3 : \leftarrow R(x, y, z), z \leq x$

then the set $\{C_1,\ C_2,\ C_3\}$ is inconsistent because there is no instance from $\mathcal{S}$ (different from empty set) that satisfies this set of constraints. Since there are no three real numbers x, y, z such that $x < y, y < z$ and $z < x$.

In the following, first, we define some measures of inconsistency degrees based on the set of measures defined in chapter 4. In the second part, we introduce a top-k algorithm, to compute the k first answers that are the most/less inconsistent answers, that is also based on the TopINC algorithm.

6.1 Fixing inconsistency problem

One way to handle inconsistency of query answers in presence of inconsistent set of constraints DC is to consider the set of maximal consistent subsets of constraints from DC. The inconsistencies of tuples (in database instance) or query answers are computed w.r.t a maximal consistent subset of DC. We denote by $MC(DC)$ the set of maximal consistent subsets of a set of constraints DC. We define, in the following, formally this notion of set of maximal consistent subsets of constraints.

$$MC(DC) = \{S \subseteq DC : S \neq \emptyset \text{ and } S \models DC \text{ and } \forall S_1, S \subset S_1 \subseteq DC \Rightarrow S_1 \not\models DC\ \}$$

In this thesis, we assume that this set exists and it is finite for any inconsistent set of constraints considered. We do not study the computation problem of $MC(DC)$. The problem of satisfiability of set of denial constraints (i.e, checking if a set of denial constraints is consistent) is studied in [16].

Example 6.2. *Consider the set of constraints considered in the example 6.1. The set of maximal consistent subsets of DC is :* $MC(DC) = \{S_1,\ S_2,\ S_3\}$ *where* $S_1 = \{C_1, C_2\}$, $S_2 = \{C_1, C_2\}$, $S_3 = \{C_2, C_3\}$.

6.1.1 Strong Consistent Query Answers (SCQA)

Given a database D and a set of denial constraints DC. If DC is inconsistent, it is not possible to find a repair of D. Since by definition an inconsistent set of constraints DC is a set of constraints for which there is no non empty instance that satisfies it. Hence, it is impossible to handle inconsistency by repairing data as done in literature [22]. As we can not define the notion of repair in the context of inconsistent set of constraints, what about the CQA [22] handling ? As the notion of repair does not exist, so the notion of CQA also does not exist.

In this section, we define a new semantic of query evaluation, based on the idea of CQA, in a context of inconsistent set of denial constraints. This notion, called strong consistent query answers, is based on $MC(DC)$. For each maximal subset of consistent constraints, we compute the consistent query answers using one of semantic of repair defined in literature. Once, these consistent query answers are computed for each $M \in MC(DC)$, the strong consistent query answers is obtained by doing intersection these consistent query answers. The strong consistent query answers is useful since it means that no matter the maximal consistent set of denial constraint from DC, these answers can be obtained.

Definition 6.1 (SCQA)**.** *Let* $DC, Q, \mathcal{I}$ *be a set of constraints, a query and a database instance on the same schema, respectively. The strong consistent query answers of* Q*, denoted* $SCQA(DC, Q, \mathcal{I})$*, is defined as follows :*

$$SCQA(DC, Q, \mathcal{I}) = \bigcap_{S \in MC(DC)} CQA(S, Q, \mathcal{I})$$

with $CQA(S, Q, \mathcal{I})$ *the consistent query answers of* Q *(with any semantic of database repair) over* $\mathcal{I}$ *on which the set of constraints* S *are applied.*

Example 6.3. *Consider the maximal consistent set of the previous example 6.2 (*$S_1,\ S_2,\ S_3$*). Consider the following instance with the query* Q*. We consider the deletion repair semantic [9].*

R

A	***B***	***C***
10	9	6
3	2	6
5	7	6
5	7	8

$Q(z) : -R(x, y, z)$

The repair according to the maximal subsets $\{S_1, S_2, S_3\}$ *are the following*

- *For* S_1 *we have* $\{\langle 10, 9, 6\rangle\}$ *So* $CQA(S_1, Q, \mathcal{I}) = \langle 6\rangle$
- *For* S_2 *we have* $\{\langle 3, 2, 6\rangle,\ \langle 5, 7, 8\rangle\}$ *So* $CQA(S_2, Q, \mathcal{I}) = \{\langle 6\rangle, \langle 8\rangle\}$

- *For S_3 we have* $\{\langle 5,7,6\rangle,\ \langle 5,7,8\rangle\}$ *So* $CQA(S_3, Q, \mathcal{I}) = \{\langle 6\rangle,\ \langle 8\rangle\}$

So, the strong consistent query answers of Q evaluated over $\mathcal{I}$ in presence of the set of constraint DC is $SCQA(DC, Q, \mathcal{I}) = \{\langle 6\rangle\}$.

The main challenge behind SCQA is its computation problem. Even the special case where $|MC(DC)| = 1$, the computation problem of SCQA remains intractable(in data complexity) in general [22]. Also, there exists some particular cases where SCQA is always empty as shown in theorem 6.1.

Theorem 6.1. *Let DC be an inconsistent set of constraints and let $\mathcal{I}$ be a database instance. For any full conjunctive query Q, we have:*

$$SCQA(DC, Q, \mathcal{I}) = \emptyset$$

Proof. Let $Q(X) : -R_1(X_1), \ldots, R_m(X_m), \phi(X_1, \ldots, X_m)$, DC, $\mathcal{I}$ be respectively a full conjunctive query, a set of denial constraints and a database instance. As the query Q is a full conjunctive query then each answer A corresponds to an unique set $\{R_1(A_1), \ldots, R_m(A_m)\}$. Assume now that $SCQA(DC, Q, \mathcal{I}) \neq \emptyset$ and assume that $A \in SQCA(DC, Q, \mathcal{I})$. This means that for each $S \in MC(DC)$ there exists a repair R that contains at least the elements from $\{R_1(A_1), \ldots, R_m(A_m)\}$. As this repair exists for each $S \in MC(DC)$ then $\bigcap_{S \in MC(DC)} S = DC$ is satisfied at least by the set $\{R_1(A_1), \ldots, R_m(A_m)\}$. Absurd since DC is inconsistent. □

We can consider the computation problem of SCQA by maximal consistent subset of DC. Given a maximal consistent subset S of DC, we use the classical techniques [23] to compute the CQA w.r.t S. So, SCQA can be computed with the same data complexity than the existing algorithms to compute CQA [23].

6.1.2 Quantifying inconsistency degrees of query answers

As preliminary work, we define below inconsistency degrees of query answers under bag semantics in a context where the set of denial constraints is inconsistent. The idea behind these new measures is to quantify inconsistency of a query answer t for each maximal consistent set of constraints. And then, we have to use an aggregate function to aggregate these different inconsistency degrees to obtain a global inconsistency degrees of t. We call this type of measure a relative measure.

Definition 6.2 (Relative measure)**.** *Let Q be a conjunctive query, DC be a set of denial constraints and $\mathcal{I}$ be a database instance. Let $\alpha \in \{CBS, CBM, TBS, TBM\}$; let $f : \mathbb{R}^p \to \mathbb{R}$ with $p \in \mathbb{N}^*$, an aggregate function. The relative measure of inconsistency degrees for an answer $t \in Q(\mathcal{I})$, based on α and f, is formally defined as follows*

$$R^f_\alpha(t, Q, \mathcal{I}, DC) = f(\alpha(t, Q, \mathcal{I}, S_1), \ldots, \alpha(t, Q, \mathcal{I}, S_p))$$

with $MC(DC) = \{S_1, \ldots, S_p\}$.

Example 6.4. *Based on the previous example (example 6.3). The database annotated is the following $\mathcal{I}$*

R

A	B	C	
10	9	6	C_3
3	2	6	C_2
5	7	6	C_1
5	7	8	C_1

Consider the query $Q_1(x,y,z) : -R(x,y,z)$ *and the measure of inconsistency* CBS*. Consider the answers* $t_1 = \langle 10,9,6\rangle$.

- $CBS(t_1, Q_1, \mathcal{I}, S_1) = 0$
- $CBS(t_1, Q_1, \mathcal{I}, S_2) = 0$
- $CBS(t_1, Q_1, \mathcal{I}, S_3) = 1$
- *Consider the aggregate function sum then* $R^{sum}_{CBS}(t_1, Q, \mathcal{I}, DC) = sum(0,0,1) = 1$
- *With the aggregate function min then* $R^{min}_{CBS}(t_1, Q, \mathcal{I}, DC) = min(0,0,1) = 0$

6.2 Query answers ranking by inconsistency degrees

In this section, we show how to modify TopINC algorithm to work with the relative measures R^f_{CBS}.

6.2.1 Top-k algorithm with R^f_{CBS}

The algorithm designed in this section considers only the relative measure R^f_{CBS}, i.e, the relative measure that is based on CBS measure and any other aggregate function f. It is denoted by RTopINC. The algorithm RTopINC has the same data complexity than TopINC. In the rest of this chapter, we consider DC the set of denial constraints and we assume that $MC(DC) = \{S_1, \ldots, S_p\}$. We use f as an aggregate function.

With the algorithm RTopINC, we proceed in two steps. The first step consists to build an index. This index, denoted $orderVio(DC, f)$, is computed once, i.e, it is independent from input query. In the second step, the top-k query is evaluated by the algorithm TopINC using $orderVio(DC, f)$.

Building of orderVio(DC, f)

We define formally $OCBS(S, f, DC)$, where $S \subseteq DC$, as follows:

$$OCBS(S, f, DC) = f(|S \cap S_1|, \ldots, |S \cap S_p|)$$

Informally, we define $orderVio(DC, f)$ as list of elements of 2^{DC}(i.e, the power set of DC). Elements of $orderVio(DC, f)$ are ordered w.r.t measure R^f_{CBS}. Formally $orderVio(DC, f)$ is defined as follows

$$orderVio(DC, f) = [E_1, \ldots, E_{2^{|DC|}}],\ \forall i, j \in [1, 2^{|DC|}],\ if\ i < j$$

$$then\ OCBS(E_i, f, DC) \leq OCBS(E_j, f, DC)$$

where no E_i ($i \in 1, \ldots, Z^{|DC|}$) is repeated in $orderVio(DC, f)$. This index has exactly the same size than the algorithm TopINC.

Example 6.5. *Continuing with the previous example, we assume min as aggregate function. In the following, we compute the OCBS for each subset of constraints:*

- $OCBS(\emptyset, f, DC) = min(CBS(\emptyset \cap S_1), CBS(\emptyset \cap S_2), CBS(\emptyset \cap S_3)) = min(0,0,0) = 0$
- $OCBS(\{C_1\}, f, DC) = min(|\{C_1\} \cap S_1|, |\{C_1\} \cap S_2|, |\{C_1\} \cap S_3|) = min(1,1,0) = 0$
- $OCBS(\{C_2\}, f, DC) = min(|\{C_2\} \cap S_1|, |\{C_2\} \cap S_2|, |\{C_2\} \cap S_3|) = min(1,0,1) = 0$
- $OCBS(\{C_3\}, f, DC) = min(|\{C_3\} \cap S_1|, |\{C_3\} \cap S_2|, |\{C_3\} \cap S_3|) = min(0,1,1) = 0$
- $OCBS(\{C_1, C_2\}, f, DC) = min(|\{C_1, C_2\} \cap S_1|, |\{C_1, C_2\} \cap S_2|, |\{C_1, C_2\} \cap S_3|) = min(2,1,1) = 1$
- $OCBS(\{C_1, C_3\}, f, DC) = min(|\{C_1, C_3\} \cap S_1|, |\{C_1, C_3\} \cap S_2|, |\{C_1, C_3\} \cap S_3|) = min(1,2,1) = 1$
- $OCBS(\{C_2, C_3\}, f, DC) = min(|\{C_2, C_3\} \cap S_1|, |\{C_2, C_3\} \cap S_2|, |\{C_2, C_3\} \cap S_3|) = min(1,1,2) = 1$
- $OCBS(\{C_1, C_2, C_3\}, f, DC) = min(|\{C_1, C_2, C_3\} \cap S_1|, |\{C_1, C_2, C_3\} \cap S_2|, |\{C_1, C_2, C_3\} \cap S_3|) = min(2,2,2) = 2$

So $orderVio(DC, min)$ *is the following*

$$orderVio(DC, f) = [\emptyset, \{C_1\}, \{C_2\}, \{C_3\}, \{C_1, C_2\}, \{C_1, C_3\}, \{C_2, C_3\}, \{C_1, C_2, C_3\}]$$

Algorithm RTopINC

The RTopINC takes three parameters: an integer k, a query $Q(X) : -R_1(X_1), \ldots, R_m(X_m), \phi(X_1, \ldots, X_m)$ and an annotated instance $\mathcal{I}$. This algorithm assumes the availability of the structure $orderVio(DC, f)$. The RTopINC algorithm is presented in algorithm 8. The RTopINC algorithm iterates on the elements in the list $orderVio(DC, f)$. For each element S in $orderVio(DC, f)$, RTopINC looks for the set of answers that violate exactly S. The array $ind(\mathcal{I}(R))$ in algorithm 8 contains the set of possible violated subsets of constraints by tuples in R. By $R_1[E_1]$ in algorithm 8, we mean all the tuples that violate exactly E_1.

As one can easily note, the algorithm RTopINC is optimal in term of the number of tuples read on disk and in the class of semi-blind algorithms.

Theorem 6.2. *For any instance* $\mathcal{I}$ *and any top-k conjunctive query* $Q^{k,R^f_{CBS}}$*, we have:*

$$cost^{\nabla}(RTopINC, Q, \mathcal{I}) \leq cost^{\nabla}(AL, Q, \mathcal{I}), \forall AL \in SBA$$

6.3 Conclusion

This chapter has presented preliminary results regarding the problem of inconsistency in relational database in the context of a set of inconsistent set of constraints. We have generalized the notion of consistent query answers to strong consistent query answers.

Algorithm 8: RTopINC

Input : $\mathcal{I}$: *database instance*, k : *integer*, Q

Output: *Res* : the k most consistent/inconsistent answers w.r.t. R^{f}_{CBS}

```
1  Res := [] of max size equal to k;
2  i := 1 ;
3  for i ≤ 2^|DC| do
4      S := orderVio(DC, f)[i];
5      for comb = (E_1, ..., E_m) ∈ ind(I(R_1)) × ... × ind(I(R_m)) do
6          if ⋃_{i=1}^{m} E_i = S then
7              tempAns := Q({I(R_1[E_1]), ..., I(R_m[E_m])});
8              add tempAns in Res ;
9              if |Res| ≥ k then
10                 return Res;
11     i := i + 1;
12 return Res;
```

We have introduced a set of new inconsistency measures quantifying inconsistent in this context. We designed an algorithm of top-k for one of these measures of inconsistency degrees. The next chapter empirically evaluates our approach and algorithms of top-k developed in this thesis.

Chapter 7

Empirical Evaluation

In this chapter, we are going to show efficiency of our approach by empirically evaluate the different algorithms. First, we show the feasibility of our measures in practice with many queries in many different datasets. In the second part, we evaluate efficiency of our top-k algorithms.

During all experiment, we use the following datasets. The most large of these datasets are real-world datasets, only few of these datasets are synthetic datasets. *Stock, this dataset contains the market data of the United States. It mainly informs about the market trend on a time interval, i.e. the minimum price, the maximum price, the initial selling price of the interval period and the end price of the market of the same interval period. It also specifies the sales quantity of the market on the period. Hospital,* registered a set of health data about admissions in many hospitals. *Food*, this dataset registered the set of food inspections done, mainly in New York city, in restaurants. *Adult*, contains data about some adult and their marital status in some countries in world (USA, India, Iran, ...). *Tax,* is a synthetic dataset. An other synthetic dataset, that we call synthetic is added, this dataset is randomly generated. Some complementary information about these datasets are displayed in table 7.1. In Table 7.1, the column *Inc* denotes the percentage of inconsistency per relation, whereas #*Tup* and #*Rel* denote the number of tuples and relations in the dataset, respectively. Finally, #*Cons* indicates the number of denial constraints per dataset. The column #*atom* gives interval of number of atoms for the constraints of a given dataset. The column *Syn* indicates the type of dataset (synthetic or not). All the queries used are described as follows: Q_1 to Q_5 are on the Hospital dataset and they contain one join; Q_6 to Q_9 are on the Tax dataset and they contain one join; Q_{10} to Q_{14} are on the synthetic dataset; Q_{10}, Q_{11} have a join across three tables, Q_{12}, Q_{13} have a join across four tables and Q_{14} has a join across five tables.

We have implemented our framework in *PostgreSQL 10* by leveraging*PLSQL* and *JDK 11*. All the experiments have been executed on a DELL Core i7 2.5 GHz laptop with 16 GB RAM running Linux OS.

Our approach of treatment of inconsistency degrees suppose existing of a set of denial constraints and availability of set of conjunctive queries for each dataset. So, we used Metanome [1], that is a tools allowing generation of a set of denial constraints from a dataset. Metanome implements a set of algorithm of discovery of constraints (functional dependency constraints and general denial constraints) developed by many works [37, 126]. Also, we use denial constraints generated by Metanome to convert into a of conjunctive queries that are used as set of conjunctive queries.

Dataset	Syn	#Rel	#Tup	#Cons		#atom	Inc(%)
				#Equal	#(In-)equal		
Stock	✗	1	244992	1	9	[1, 2]	18.62
Hospital	✗	1	114919	3	39	2	100
Tax	✓	1	99999	1	49	2	100
Synthetic	✓	5	1012524	6	9	[1, 3]	89.34
FoodInspection	✗	1	204896	N/A			
Adult	✗	1	48842				
*Rand*1	✓	4	1000000				

Table 7.1: Datasets used in our empirical evaluation.

7.1 Measures Evaluation

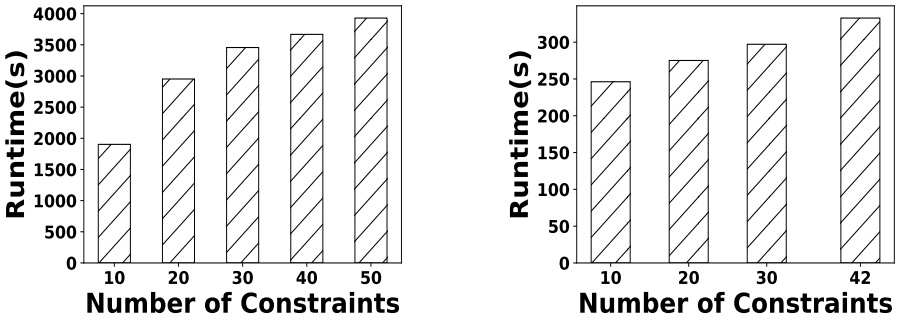
This section evaluate our measures, we evaluate the running time of the annotation phase, the running time to compute each measure and a qualitative study.

7.1.1 Database Annotation

This preprocessing of data can take more running time. It is done once, only when there are some update in database. The approach that consists to do that once is interesting in the case of data warehouse. Because, in data warehouse, there are less updates. So, in the following, we evaluate our approach, begin with the conversion of the set of denial constraints into conjunctive, the evaluation of these converted denial constraints to compute the why-provenance until the annotation phase of each tuple in the database. Before annotation four columns, *vioset*, *viobag*, *inc* and *incbag* containing the set of violated constraints in binary format for set answers semantics measures with constraint-based approach, the number of violated constraints for bag semantics answers with constraint-based approach, a monomial (either 1 or identifier of the tuple) for set semantics answers and tuple-base approach and 1 either 0 for bag semantics answers and tuple-based approach, respectively, are added on each relation. Once, the why-provenances of each obtained conjunctive query is computed, each tuple in the instance is annotated through its four columns. This empirical evaluation is done only for STOCK, HOSPITAL, TAX, SYNTHETIC data since these databases contain more large set of denial constraints.

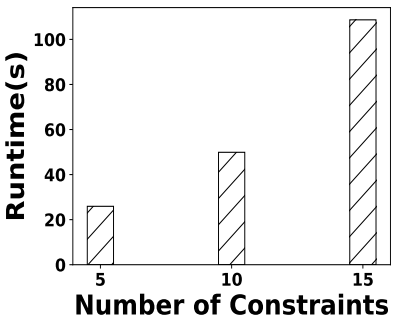
Figure 7.1 shows the runtimes for each dataset while varying the number of constraints. We can observe that the runtimes of the *K-instance* transformation linearly scale with the number of constraints for all datasets. They range between tens and thousands of seconds, depending on the dataset. We observed the highest runtimes only with one dataset (Tax), which has fifty denial constraints and took approximately 1h to transform 100000 tuples. Such a transformation is part of the pre-processing and only done one time for the annotated instances, thus it remains quite reasonable.

One can also notice that there is a huge gap between runtimes of transformation of Tax (Figure 7.1.a) and Hospital (Figure 7.1.b) despite the fact that these two datasets have similar characteristics. The observed gap is due to the fact that the constraints in the dataset Tax are more sophisticated than the constraints in the dataset Hospital (i.e., with larger built-in atoms).

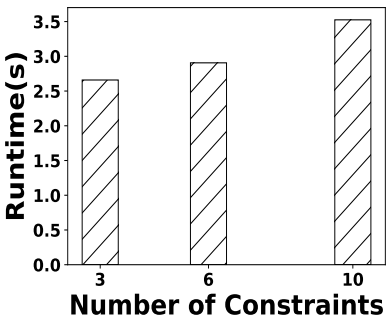


(a) Tax

(b) Hospital



(c) Synthetic



(d) Pstock

Figure 7.1: Transformation of an instance into a $\mathbb{N}[Y \cup \Gamma]$-instance

	Query	#Answers	CBM	CBS
Q_1	1 *s*	2 891 897	119 *s*	59 *s*
Q_2	214 *ms*	560 161	22 *s*	11 *s*
Q_3	41 *ms*	114 919	2 *s*	2 *s*
Q_4	31*ms*	50	3 *ms*	2 *ms*
Q_5	57 *ms*	625	31 *ms*	19 *ms*
Q_6	23 *ms*	99 999	546 *ms*	546 *ms*
Q_7	97 *ms*	100 777	903*s*	558 *ms*
Q_8	518 *ms*	488 505	6 *s*	4 *s*
Q_9	61 *ms*	303	83 *ms*	80 *ms*
Q_{10}	231 *ms*	582	12 *ms*	12 *ms*
Q_{11}	472 *ms*	505 046	7 *s*	4 *s*
Q_{12}	3 *s*	14831052	5 *mn*	2 *mn*
Q_{13}	1 *s*	7 384 207	2 *mn*	1 *mn*
Q_{14}	938 *ms*	287 242	9 *s*	3 *s*

Across all tuples in the query output
(a)

	CBM	CBS
Q_1	41 μs	20 μs
Q_2	39 μs	19 μs
Q_3	17 μs	17 μs
Q_4	60 μs	40 μs
Q_5	49 μs	30 μs
Q_6	5 μs	5 μs
Q_7	9 μs	5 μs
Q_8	12 μs	8 μs
Q_9	273 μs	264 μs
Q_{10}	20 μs	20 μs
Q_{11}	13 μs	7 μs
Q_{12}	20 μs	8 μs
Q_{13}	16 μs	8 μs
Q_{14}	31 μs	10 μs

One tuple at a time
(b)

Figure 7.2: *CBS* and *CBM* computation overhead.

7.1.2 Measures Computation

In order to gauge the overhead of running a query Q with inconsistency degrees, we have employed our 14 queries and measured the overhead per each tuple in the answer (Table 7.2.a) as well as the total overhead for the complete output of the query (Table 7.2.b). The obtained results are reported in Table 7.2. The greater is the size of the output of a query the larger is the overhead of query execution with inconsistency degrees. The columns *query and #answers* in Table 7.2.a are the total query runtime of the original query on an inconsistency-free database instance and the size of query answer, respectively. Depending to the size of output of the queries, the overhead ranges between 2*ms* and 6*m*, respectively for queries Q_4 and Q_{12}. (respectively, yellow and red cells in Table 7.2.a). The difference can be explained by looking at the size of the answer set of Q_4 and Q_1 that are 50 tuples and 15M tuples, respectively. These overhead are, however, not trustworthy to understand the overhead of our approach, since they concern the entire output set of queries, whereas our algorithms (developed in chapter 5) returns the top-k results tuple by tuple. Thus, one should look at the overhead per tuple in Table 7.2.b. We can observe that the overheads per tuple are reasonable in all cases, and range between 5 microseconds and 293 microseconds (yellow and red cells in Table 7.2.b, respectively).

7.1.3 Qualitative Study

We have designed an experiment devoted to show the utility of our inconsistency measures on real-world inconsistent data. We chose two real-life datasets, namely Adult, used and containing census data, along with and Food Inspection including information about inspections of food establishments in Chicago. The features of the two datasets are shown in Table 7.1 Table 7.2.b reports the constraints of Adult, namely A_1 and A_2, that have been derived using Holoclean [128]. While A_1 indicates that men who have 'married' as marital status are husbands, A_2 expresses the dual constraint for women.

A_1:$\leftarrow$ Adult(A,MS,Re,S,...)$\wedge$ S='Female' $\wedge$ Re='Husband'
A_2:$\leftarrow$ Adult(A,MS,Re,S,...) $\wedge$ S='Male' $\wedge$ Re='Wife'
A_3:$\leftarrow$ Adult(A,MS,Re,S,...) $\wedge$ Re='Husband' $\wedge$ MS='Marr-civ-sp.'

(b) Constraints on Adult.

F_1: $\leftarrow$ Inspection(I_1,FT_1, V_1, Re_1, L_1, ...) $\wedge$ $L_1 = L_2$ $\wedge$ Inspection(I_2,FT_2, V_2, Re_2, L_2, ...) $\wedge$ $N_1 \neq N_2$
F_2: $\leftarrow$ Inspection(I_1,FT_1, V_1, R_1, L_1, ...) $\wedge$ Inspection(I_2,FT_2, V_2, R_2, L_2, ...) $\wedge$ $L_1 = L_2 \wedge R_1 \neq R_2$
F_3: $\leftarrow$ Inspection(I_1,FT_1, V_1, R_1, L_1, D_1, ...) $\wedge$ Inspection(I_2,FT_2, V_2, R_2, L_2, D_2, ...) $\wedge$ $IT_1 =' consultation' \wedge IT_2 \neq' consultation' \wedge D_2 < D_1$

(c) Constraints on Food Inspection.

AQ1:SELECT * FROM adult a1, adult a2 WHERE a1.sex = 'Male' AND a2.sex = 'Female'AND a1.country = a2.country AND a1.income = a2.income
FQ1: Select t2.license From inspection t1, inspection t2, inspection t3 where (t1.results = 'Fail' or t1.violations like '%food and non-food contact %') and t1.license=t2.license and t1.license=t3.license and t2.results $\neq$ 'Fail' and t2.inspection_type = 'Canvass' and t3.inspection_type='Complaint' and t1.inspection_date<t3.inspection_date and t3.inspection_date <t2.inspection_date and t1.zip >= 60666 and t2.zip > 60655 and t3.zip > 60655

(d) Queries on Adult and Food Inspection.

Table 7.2: Real-world datasets with their denial constraints.

In addition, we handcrafted a third constraint A_3 establishing that adults who are not in a family should not have 'married' as marital status. This third constraint allows to capture violated tuples that overlap with the tuples violated by the two former constraints. We also built meaningful denial constraints for the second dataset as shown in Table 7.2.c. The constraint F_1 (respectively, F_2) states that a licence number, which is a unique number assigned to an establishment, uniquely identifies the legal name of the establishment (respectively, its *risk category*). The constraint F_3 states that if a given establishment has been inspected for *'consultation'* at a date d, one cannot expect to have an inspection of a different type prior to d for the same establishment. This is because the attribute Inspection type takes the value *'consultation'* when the inspection *"is done at the request of the owner prior to the opening of the establishment."*

We report in Table 7.2.d the considered queries for the two datasets. The query AQ1 on Adult finds all couples of male and female living in the same country and earning the same income. The query FQ1 on Food Inspection retrieves the licenses of establishments in a specific area that were subject to three inspections: the first one having either a failing inspection or a violation related to *"food and non-food contact surface"*, followed by an inspection issued as a response to a complaint and then a non-failing normal inspection.

Table 7.3.a. shows the violations of constraint for the two datasets.

We can notice that there are different kinds of tuples returned by $AQ1$ as illustrated in Table 7.3.b. The majority of the results ($276M$ tuples) are consistent, thus both CBS and CBM are equal to 0, while the remaining answers exhibit 1 or 2 as inconsistency degrees. Most of the inconsistent tuples violate one constraint at a time (in the order A_3, A_1 and A_2) while the rest of the tuples violate two constraints. For this dataset, in the majority of the cases a constraint is violated at most once and hence CBS and CBM

measures are not discriminating, except for the 23 answers where violations occur twice (5th line of Table 7.3.b). These tuples are captured when running TopINC for top-100 tuples starting from the most inconsistent ones as illustrated in Table 7.3.c. Note that, while CBM does not distinguish between the 71 most inconsistent answers of $AQ1$ (answers with $CBM = 2$ corresponding to the first four rows of Table 7.3.c), the CBS measure provides a different ranking for the 23 answers of the 4th row of this Table.

We show in Table 7.3.d the inconsistency degrees of the answers when evaluating query FQ1 on the second dataset. Note that for this query the CBS degrees vary from 0 up to 3 while the CBM degrees range from 0 up to 9. We observe now that many answers are not distinguishable under CBS while they exhibit a wider range of CBM degrees (e.g., CBM varies from 1 to 3 for answers having $CBS = 1$). This again shows that CBS and CBM provide the user with two different types of information, both being useful to carry out the ranking.

Finally, we show by means of examples the remarkable difference between our approach and CQA[1]. Note that as we already pinpointed in chapter 4 (lemma 4.2), these are complementary approaches. Table 7.3.e shows three CQA-consistent answers for query $FQ1$. First, we note that all our consistent answers (i.e., with $CBS = CBM = 0$) are also CQA-consistent as stated in Lemma 1. The converse is not true as it can be observed in Table 7.3.e where the answer $\langle 34183 \rangle$ is CQA-consistent but not consistent in our framework (with CBS and $CBM \neq 0$). On another note, we can notice that the CQA approach does not distinguish between the three CQA-consistent answers of Table 7.3.e. In particular, the information that the CQA-consistent answer $\langle 34183 \rangle$ is computed using inconsistent base tuples (violation of F_1) is not conveyed by CQA.

7.2 Top-k Algorithms Evaluation

This section is dedicated to the evaluation of some algorithms of top-k developed in chapter 5.

7.2.1 TopINC Performance vs. Baseline.

We have implemented a baseline algorithm leveraging PostgreSQL, where all answers of a query are computed beforehand and then sorted (ORDER BY) and filtered (LIMIT k). Figure 7.3 shows the performance of TopINC (with k varying from 10 to 300) as opposed to the aforementioned baseline algorithm. We have chosen five queries as representatives of different datasets and join sizes ranging from one join (Q_1,Q_2, Q_8) and three joins (Q_{11}) to five joins (Q_{14}). The algorithm TopINC (blue bar) can be up to 2^8 times faster than the baseline approach as shown in Figure 7.3.a. There is only one query, i.e. the most complex query Q_{14}, for which TopInc has lower performance compared to the baseline, starting from a value of $k \geq 200$. The reason for that is the fact that TopInc for higher values of k and higher number of joins in the query will inspect more buckets and try to perform more joins that will likely produce no answers. Furthermore, notice that in all these experiments the baseline turns to have advantageous with respect to TopInc, as it leverages the query planning of Postgres and opportunistically picks the most efficient join algorithm. Despite these advantages, our approach is still superior in terms of performance in the majority of the cases.

[1]We consider repair by deletion and symmetric set difference as measure of minimality [9].

In Figure 7.4, from 7.4.f to 7.4.j, we measure the memory consumption of our approach for the same queries. We observe that TopINC always consumes less memory than the baseline.

We also ran another experiment on synthetic datasets to study the impact of other parameters on the performance of TopINC. The results are reported in Figure 7.5.

Precisely, we wanted to study the dependency of TopINC on the following parameters: the number of answers to be returned (k), the number of violated constraints (DC), the size of search space formed by DC (i.e, the number of subsets of constraints violated by base tuples) and the exact size of output of Q (i.e, $|Q(\mathcal{I})|$).

In these results, we focused on a relatively simple query Q containing a join between two synthetic relations (of size 1000 each one) and we kept constant the value of k (equal to 20). We ran this experiment with varying number of denial constraints DC from 10 to 30, respectively in Figures 7.5.a, 7.5.b and 7.5.c. In each of these plots, we vary the selectivities of Q and the size of the search space of the algorithm. One can see that TopINC outperforms the baseline in all cases and is particularly advantageous with larger query outputs and smaller search space.. The underlying reason is that the greater is the output size of the query, the larger is the probability to find answers within the first combinations scanned within the search space.

7.2.2 TupIncRank Performance

In this section, we compare the TupIncRank algorithm to the naive algorithm, the baseline algorithm and the rankJoin algorithm developed in [84], that is one of the best top join algorithms. We compare the running time of rankJoin to our TupIncRank since this last one works for monotone function TBM and TBS (in the case where the query is a free self join conjunctive query).

In this last experiment, we consider the database $Rand1$ that is a synthetic database. The $Rand1$ database is composed to four relations generated randomly. No set of denial constraints is considered in this database, so we randomly associate to each 1 or 0 as annotation (in colum *incbag*) to specify its inconsistency nature. As the TupIncRank algorithm, that is designed to the measures TBM and TBS, only needs to have the database annotated by 0 (consistent) or 1 (inconsistent), this nature of the database is enough. The database $Rand1$ contains one million of tuples, each relation contains 250000 tuples.

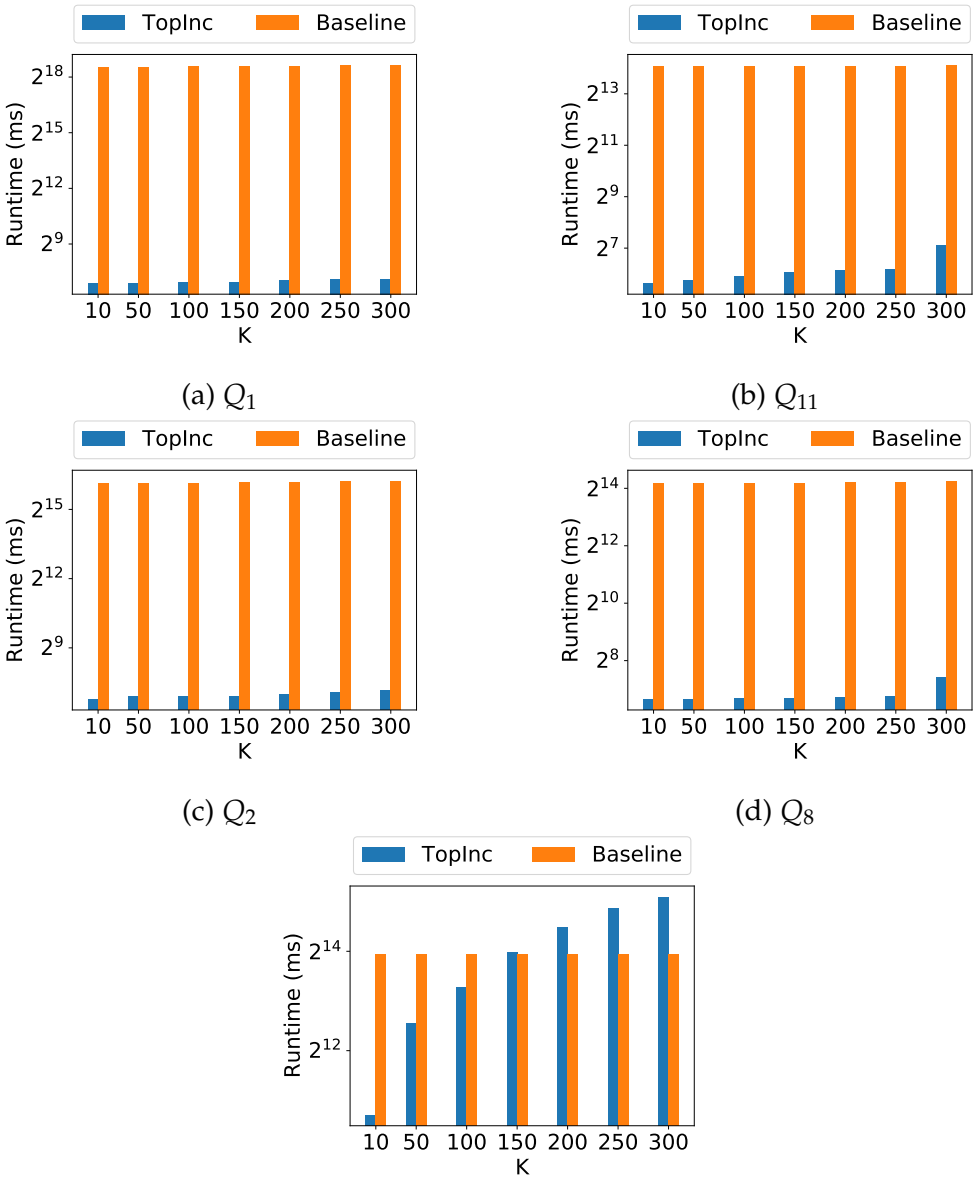
SO, first we compare the running time of these algorithms, then the number of tuples read from input relations by each algorithm after acclaiming the top-k algorithm and the number of additional answers computed by each algorithm outside the top-k answers. So, we consider the two following queries ($QX1$ and $QX2$). We vary the value of k from 20 to 200 with step equal to 20 between two consecutive value of k. This interval is randomly chosen, these algorithms remain with similar behavior with other values of k.

$$\begin{array}{|lll|}\hline QX1(X_1) & :- & R(a,b,c,d,e,f), T(a,c_1,e_1,j,f), c < c_1 \wedge b \geq j \wedge e <> e_1 \\ QX2(X_2) & :- & G(d,e,f,k), S(g,h,e,j,k_1), \wedge f <> k \\ \hline\end{array}$$

With $X_1 = a,b,c,d,e,f,c_1,e_1,j$ and $X_2 = d,e,f,k,g,h,j,k_1$.

The queries $QX1$ and $QX2$ have 1518203018 and 11175284 answers, respectively, when evaluated over $Rand1$.

The figure 7.6 shows the performance of TupIncRank algorithms compared to some top-k algorithms that work with a monotonic scoring function. The TupIncRank algorithm remains the best no matter the query. It largely exceeds in performance the

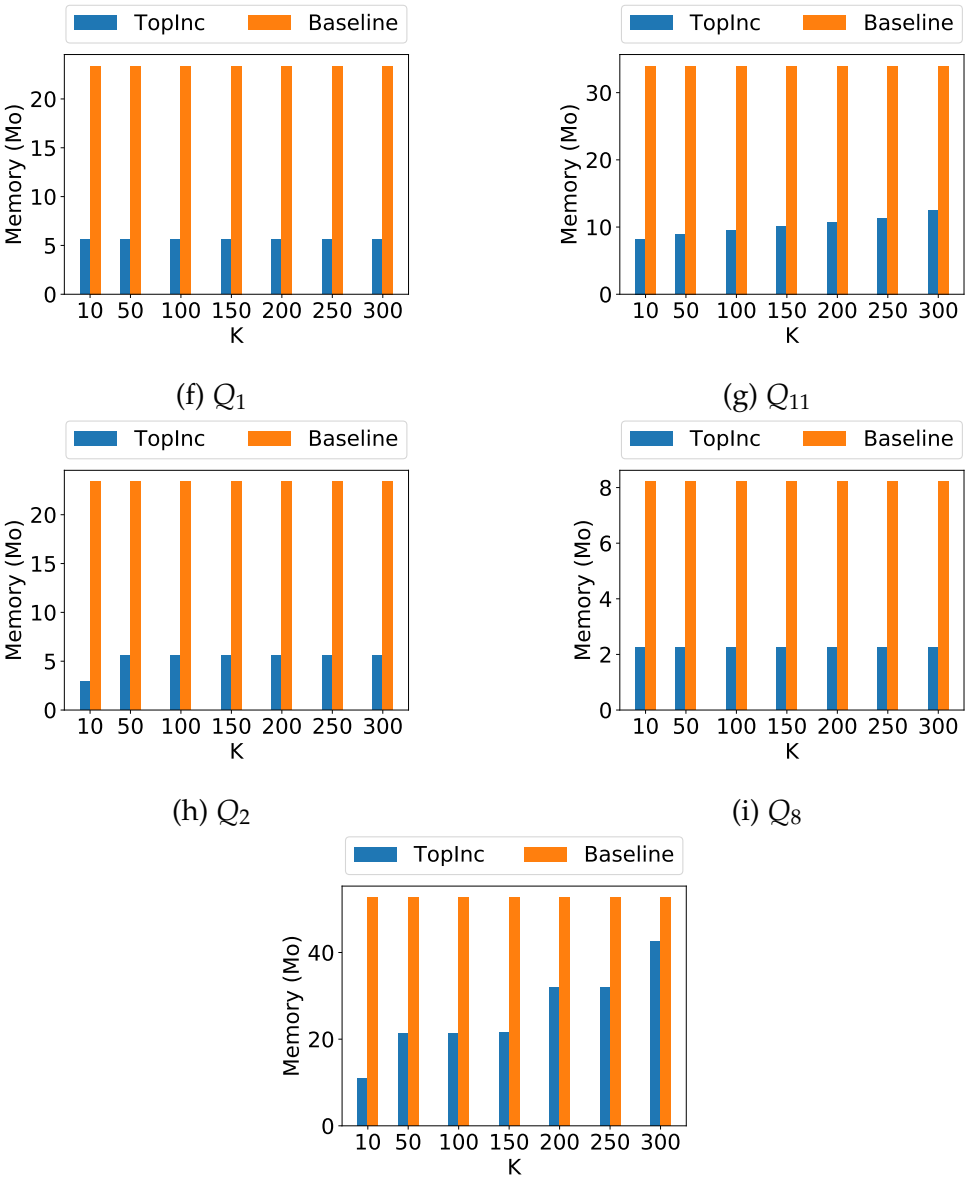


(a) Q_1 (b) Q_{11}

(c) Q_2 (d) Q_8

(e) Q_{14}

Figure 7.3: TopINC performance vs baseline ($\alpha = CBS$): Runtime



(f) Q_1 (g) Q_{11}

(h) Q_2 (i) Q_8

(j)Q_{14}

Figure 7.4: TopINC performance vs baseline ($\alpha = CBS$): Memory footprint

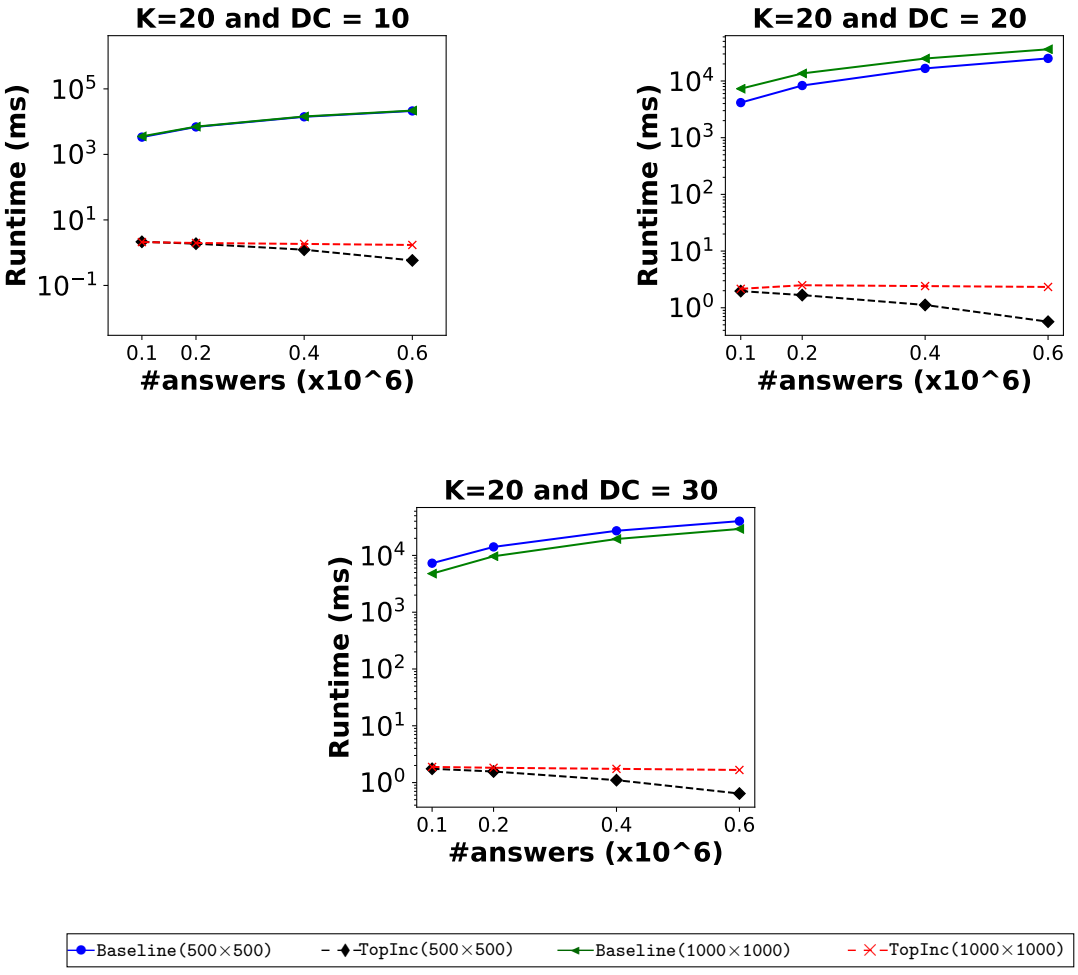
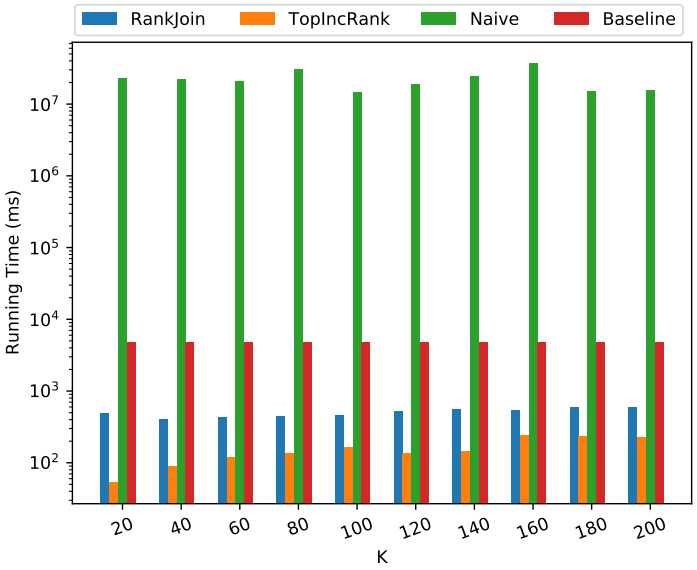
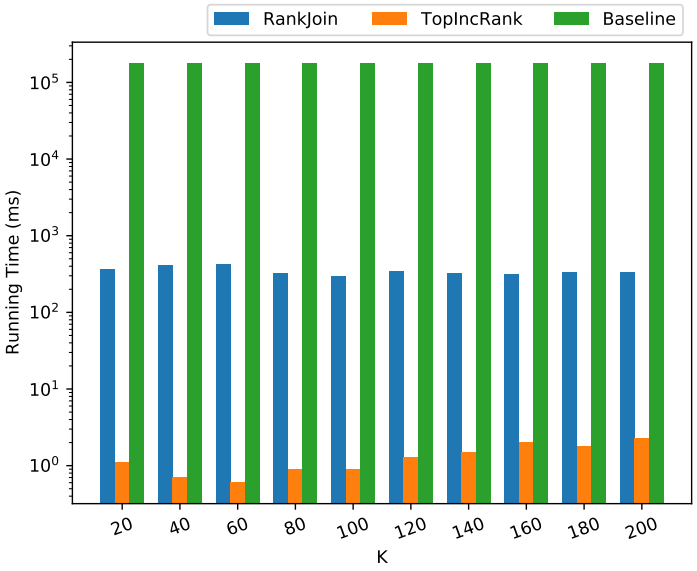


Figure 7.5: TopINC vs. Baseline ($\alpha = CBS$)



(a). Running time of $QX1$ with four top join algorithms



(b). Running time of $QX2$ with thre top join algorithms

Figure 7.6: Performance comparison between TupIncRank and other algorithms in running time

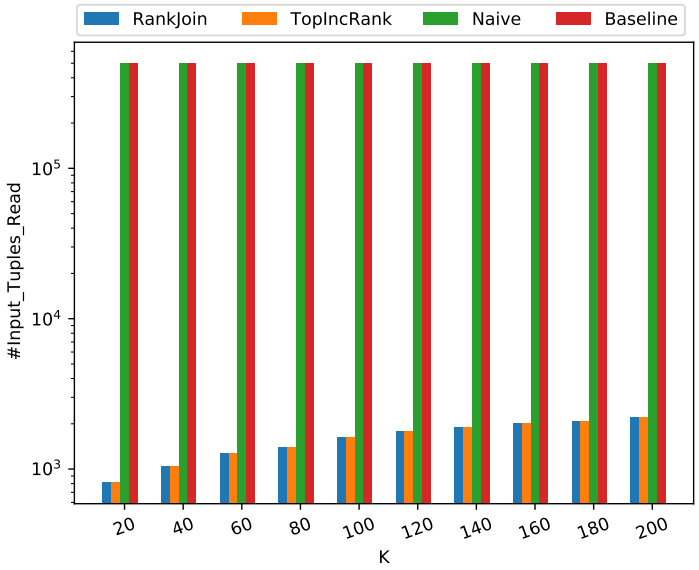
RANKJOIN algorithm that is one of the best top join algorithms in literature. The TupIncRank algorithm is best than RANKJOIN since RANKJOIN starts with a preprocessing phase that sorts first the input relations instead of the TupIncRank algorithm scans just the relation with selectivity (consistent tuples first and then inconsistent tuples). This experiment shows the non usability of the naive algorithm in practice, as one can denote the naive algorithm is not present in figure 7.6.(b) because it takes more than four days to run for ten different values of k. The running time of TupIncRank algorithm is also compared to the running time of baseline algorithm (The execution of the query in Postgresql with *ORDER BY* and *LIMIT* clauses).

The figure 7.7, we show the number of tuples read in input relations by different relations. We show that the naive algorithm reads all the input tuples before acclaiming the top-k answers. But, the RANKJOIN and TupIncRank algorithms have the same number of tuples read before acclaiming of the top-k answers. These two last algorithms read less than three thousand tuples before acclaiming of top two hundred answers instead of the naive and the baseline algorithm that read one million of tuples before acclaiming of the top-k answers.

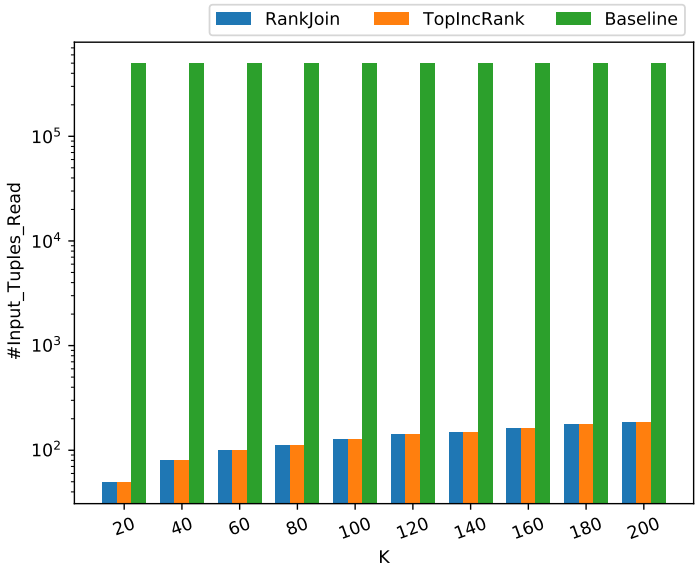
We are also interested by the number of additional answers computed outside the top-k answers. In this direction, the figure 7.8 compares the naive algorithm, RANKJOIN algorithm and TupIncRank algorithm in term of additional computed answers. Our algorithm (TupIncRank) computes always exactly only the top-k answers without any additional answer (so, the number of additional answers is always 0) opposed to the RANKJOIN algorithm that, some times, computes some few additional answers since the maximum bound remains smaller than the bound of the last of the k first answers computed, so the RANKJOIN can not stop the running otherwise the current set of answers can be incorrect (can contain answers that are not in the top-k answers, for example the last one in this set). So, these additional answers computation is unavoidable. As the figure 7.8 shows, the naive algorithm computes more than one million answers as additional set of answers.

7.3 Conclusion

We extensively evaluated our approach. First, we empirically shown the running time of the preprocessing phase. We have noticed that this step can take more but as it is done only one time, it is reasonable. In the second part, we evaluated the overhead that measures of inconsistency can add on the queries processing time. We also done a qualitative experiment of these measures and compared our approach with CQA approach. In the last section, we confronted the performance of some of our algorithms, developed in chapter 5, to the baseline algorithm (algorithm of top-k implemented in Postgresql) and naive algorithm. Our algorithms remain the best. In the next chapter, we are going to describe the new tool, called INCA, that enables profiling of inconsistency in whole of the database and the query answers.

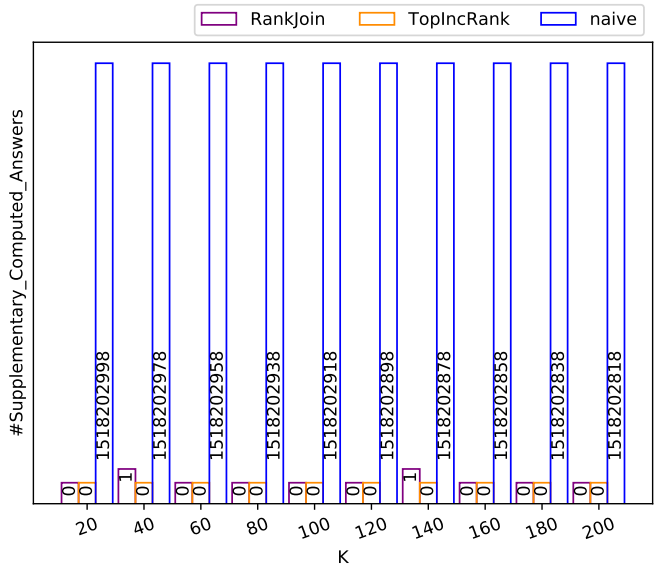


(a). Number of tuples read during running of $QX1$

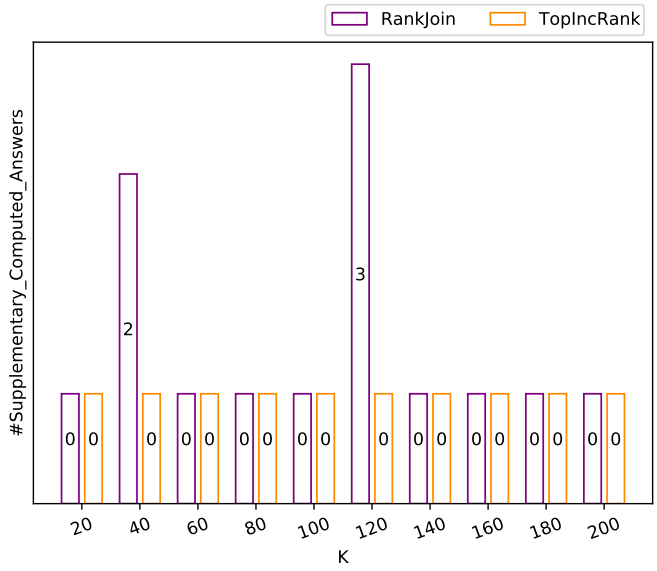


(b). Number of tuples read during running of $QX2$

Figure 7.7: Performance comparison between TupIncRank and other algorithms in memory used



(a). Additional tuples computed during performing of $QX1$ outside top-k answers



(b). Additional tuples computed during performing of $QX2$ outside top-k answers

Figure 7.8: Performance comparison between TupIncRank and other algorithms in additional computed answers

Viol. Const.	#Viol. F.Insp.	Viol. Const.	#Viol. Adult
$\emptyset$	$191K$	$\emptyset$	$48K$
F_1	7715	A_1	7
F_2	360	A_2	5
F_3	2366	A_3	23
F_2F_1	1977	A_1A_2	0
F_3F_1	181	A_1A_2	0
F_3F_2	2	A_2A_3	0
$F_3F_2F_1$	439	$A_1A_2A_3$	0

(a) Data Inconsistency.

CBS	CBM	#Ans	Annot.
0	0	$276M$	$\emptyset$
1	1	$99K$	A_1
1	1	$28K$	A_2
1	1	$13K$	A_3
1	2	23	A_3^2
2	2	10	A_1A_2
2	2	32	A_1A_3
2	2	6	A_2A_3

(b) Distrib. of AQ1 Answ.

CBS	CBM	#Ans	Annot.
2	2	32	A_1, A_3
2	2	6	A_2, A_3
2	2	10	A_1, A_2
1	2	23	A_3^2
1	1	29	A_1

(c) Top-100 AQ1 Answ.

CBS	CBM	Ans.
0	0	$\langle 1141505 \rangle$
0	0	$\langle 1042895 \rangle$
1	3	$\langle 34183 \rangle$

(e) Comparison with CQA.

CBS	CBM	#Ans	Annot.
0	0	6239	$\emptyset$
1	3	495	F_1^3
2	6	17	$F_2^3F_1^3$
1	1	6	F_3
1	2	16	F_3^2
1	3	3	F_3^3
2	4	72	$F_3F_1^3$
3	8	135	$F_3^2F_2^3F_1^3$
3	9	36	$F_1^3F_3^3F_2^3$

(d) Distrib. of FQ1 Answ.

Table 7.3: Results of the qualitative study.

Chapter 8

INCA: INCONSISTENCY-AWARE DATA PROFILING and QUERYING

As quantifying the level of inconsistency of base tuples is crucial for numerous data curation, data science and machine learning tasks, so it is useful to have a tool allowing to manage inconsistency by quantifying its degrees. INCA quantifies inconsistency of database tuples and queries answers tuples. It makes also profiling of inconsistency in database and in queries answers. Based on our knowledge, there is no tool that allows to make profiling of inconsistency.

Whereas cleaning is a prominent step of data preprocessing workflows, it might be difficult to choose the repairing parameters and values in all cases, especially when applying updates to the underlying data [11, 152] or when the underlying data is sensitive or cannot be modified in situ. It is often the case in real-world applications involving data science workflows in which domain-specific data needs to be assessed from a quality viewpoint.

In particular, in INCA, we first leverage why-provenance [27] in order to identify the inconsistent base tuples of a relational instance with respect to a set of DCs. Then, we rely on provenance polynomials [74] in order to propagate the annotations of inconsistencies from the base tuples to the answer tuples of Conjunctive Queries (CQs). Ultimately, these annotations can be used in conjunction with data cleaning steps to provide explainable results enriched with provenance-based information about inconsistency. Building upon the computed annotations, we use (choose) four measures of inconsistency degrees, which consider single and multiple violations of constraints both in the context of bag answers semantics (CBS and CBM measures) [87], and set answers semantics (CSS_{min} and CSM_{min} measures). The system INCA implements TopINC that efficiently execute top-k and threshold queries on top of the annotated data, respectively aiming at computing the top-k most (in-)consistent results and the results satisfying an inconsistency threshold. With this tool, we engage the user in the following scenarios with INCA:

- Precise statistics of the number and proportions of violations as well as the violations per constraint or per subset of constraints in a given database.
- Inconsistency-aware profiling of the underlying data encompassing inconsistency-aware data exploration by constraint and set of constraints.
- Value grouping allows to show a flexible distribution of the violations and to identify values that lead to most violations.

- Inconsistency-aware profiling of top-k and threshold queries allowing to display inconsistency degrees (according to the four aforementioned measures CBS, CBM, CSS_{min} and CSM_{min}) and violation distributions of the result tuples.

These scenarios can be seamlessly applied to any data science pipeline. Whereas a swathe of data profiling and data discovery tools exist for a variety of constraints (from functional dependencies to denial constraints) as extensively documented in the literature [3], to the best of our knowledge INCA is the first system to (1) enable inconsistency-aware data profiling leveraging the constraint-based view of the violations; (2) allow inconsistency-aware top-k and threshold query execution building on the inconsistency measures. These features pave the way to clear-cut use cases, such those underpinning explainable and selective data cleaning as well as quality-driven ranking of query results.

This chapter is organized as follows: the section 8.1, we briefly describe INCA system, section 8.2 is consecrated to the description of INCA implementation and in section 8.3 we present a user case with some database described in previous chapter.

8.1 Overview of INCA

The system INCA is in a form of a web application. It is composed from many components. In the following, we describe the architecture of INCA system.

8.1.1 Architecture of INCA System

The INCA system architecture is depicted in figure 8.1. The system INCA comes equipped with a range of methods that enable to efficiently analyze a given dataset and collect inconsistency-based statistics and information based on the inconsistency degrees computed at the preprocessing step. We describe in the sequel the main components of INCA.

- The Preprocessing module is in charge of annotating base tuples with monomial expressions encoding the constraints violated by each base tuple.
- The Data I-Profiling (Inconsistency-based Profiling) module enables a better understanding of data inconsistencies. Based on the annotations computed at the preprocessing step, it offers a set of tools to extract and aggregate information about inconsistent data. In particular, it encompasses a dashboard which provides insights into various dimensions of data inconsistencies, e.g., percentage of consistent/inconsistent data, distribution of data values in the columns that cause constraints violations, histograms of inconsistent tuples distribution w.r.t. constraints, and so forth. This module includes an I-Explore (Inconsistency-based data exploration) tool, which can be used to identify and summarize the portion of the data causing the inconsistency.
- The Query I-Profiling module allows inconsistency-aware query processing w.r.t. the four inconsistency measures defined above. In particular, it provides the following main functionalities:
  - computing query answers together with their associated inconsistency degrees,

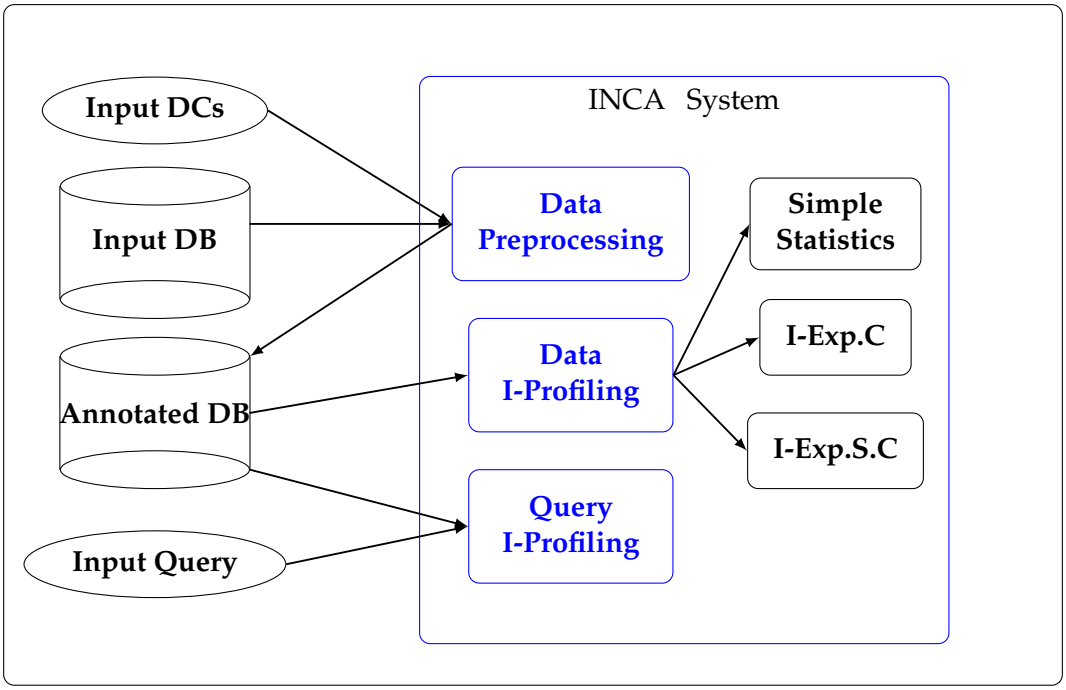


Figure 8.1: Architecture of INCA

- computing top-k answers w.r.t. inconsistency degrees (i.e., k most or less inconsistent answers),
- computing query answers having an inconsistency degree below a given threshold (threshold queries).

This module implements the TopINC algorithm and its specific index [87].

8.2 Implementation of INCA

The system INCA is implemented in JAVA. The back-end part is implemented using SpringBoot framework in an MVC structuring. The front-end is implemented in HTML, CSS and JAVASCRIPT. To describe graphically the analyze of inconsistency, we use ChartJS that is a JAVASCRIPT library to make graphical representations. As data source, we allow only Postgresql database manager.

We adopt to separately develop the useful functions as service web (in back-end by SpringBoot) and the user interface (front-end). This separation allows a quick and an easy maintenance. The complete code base of INCA is available on Github[1].

8.3 User interaction with INCA

The audience will be able to use INCA to execute the following main steps:

- identifying inconsistent base tuples using why-provenance and annotating the initial database with inconsistency degrees,
- examining data inconsistencies using I-Profile and I-Explore and

[1]https://github.com/oussissa123/INCA

- running inconsistency-aware top-k and threshold queries.

In the rest of this section, we focus on presenting the functionalities of INCA using Food Inspection, a dataset which includes information about inspections of food establishments in Chicago. The dataset Food Inspection is a long with a five meaningful denial constraints manually set up. In the following, we demonstrate step by step the diverse tools contained in INCA.

8.3.1 Annotating the initial database

Users can access INCA through a GUI. The first step is to connect to the Postgresql server and select one of the provided databases. The user can then select the constraints to be considered in the scenario from a list of predefined denial constraints. The user has the possibility to update, delete or create new denial constraints. Once an input database and its associated set of constraints have been selected, the user can launch the annotation process. As a result, each tuple in the database is annotated with a monomial, which encodes the constraints violated by the tuple. For each table, the generated annotations are stored in a new column, named prov and the number of violations are stored in column violation. Each monomial is encoded as an integer whose binary representation indicates the violated constraints.

8.3.2 Data I-profiling

In the second step of our demonstration, users will use the Data I-profiling module to analyze the annotated data and collect statistics and information about data inconsistencies. Figure 8.2 depicts the Simple statistics dashboard, which shows statistics on the number and proportions of violations as well as the violations per constraint or per subset of constraints. The «Proportion of violations» section of the dashboard informs the user about the percentage of inconsistent and consistent tuples, respectively, using a pie chart. The three other sections of the dashboard show the distribution of violations of constraints according to the input database. The board entitled «Distribution of violations» shows the percentage of tuples violating the same number of constraints. The board «Violation by Constraint» and «Violations by Subset of Constraints» expose the percentage of tuples violating the same constraint and the same subset of constraints, respectively.

Throughout the entire process, the users can decide to disable some constraints and perform the analysis only on the selected subset of the constraints.

Figure 8.3 depicts the GUI of the I-Exp.C tool that enables an inconsistency-based exploration by constraint. In the scenario, the user starts by selecting a constraint that will be analyzed. The «Constraints Correlation» section displays information about the correlation of the selected constraint with the other constraints in terms of proportion of common tuples that violate the two constraints.

The user is also assisted by a graphical query builder in order to build queries that show a flexible distribution of the violations and to identify values that lead to most violations w.r.t. the selected constraint.

The last Data I-Profiling functionality enables the user to explore subsets of constraints. The GUI of this module is depicted in Figure 8.4. After a user selects a subset of constraints to be analyzed, statistics about number of violations and number of constraints violated per each tuple are displayed.

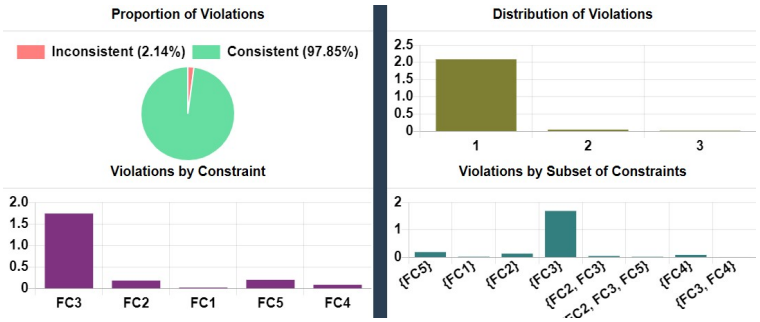


Figure 8.2: Simple statistics

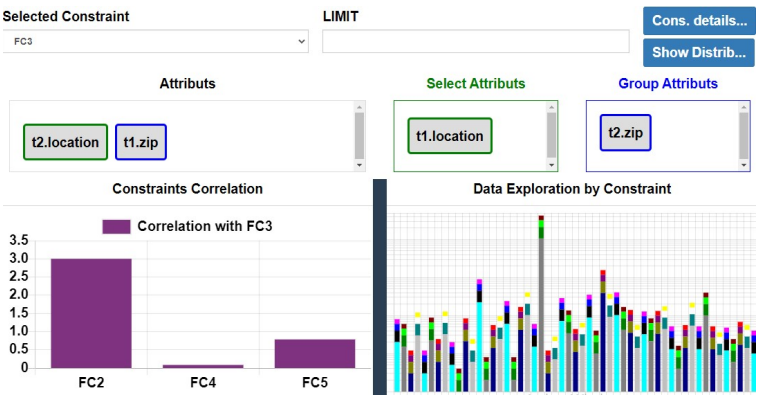


Figure 8.3: Inconsistency Exploration by Constraint

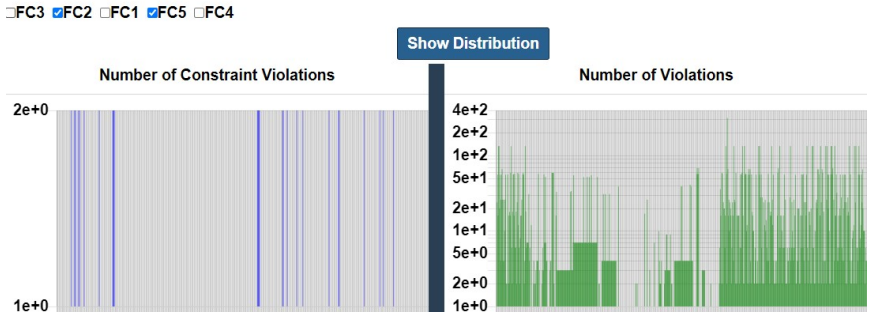


Figure 8.4: Inconsistency exploration by subset of constraints

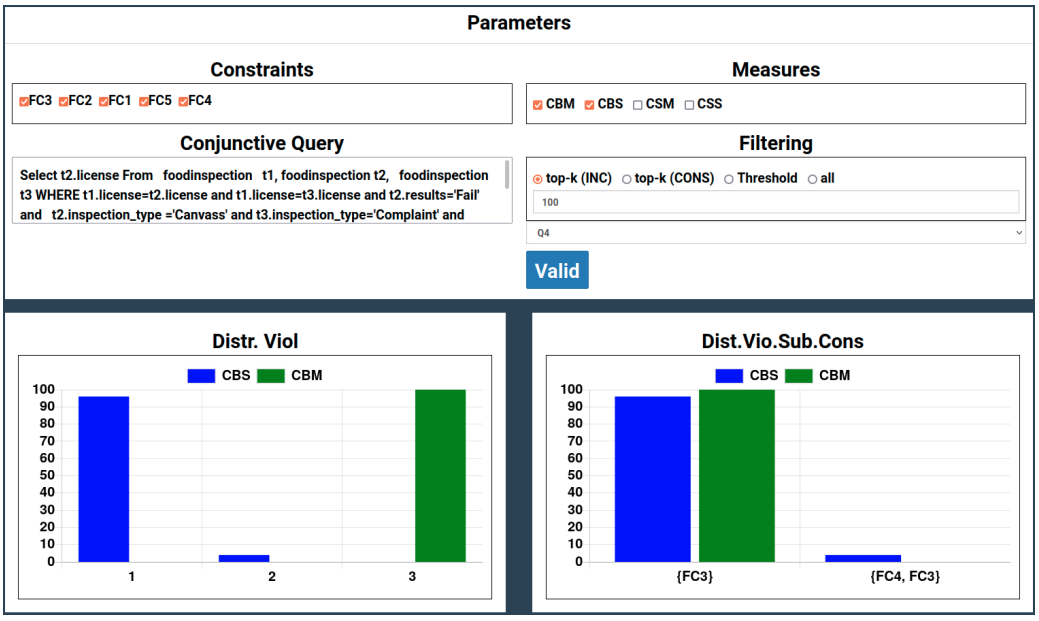


Figure 8.5: Inconsistency-aware query answering

8.3.3 Inconsistency-aware query answering

In this third step, we allow a user to explore the querying capabilities of INCA using the Query I-Profiling module. Two main classes of queries are supported by INCA:

- **Top-K query**, i.e. a conjunctive query that uses as parameters an integer k and an operator $o \in \{<, >\}$ and outputs the k *best* answers w.r.t. the considered measure α and the operator o. In the sequel, a top-k query is denoted $Q^{k,o,\alpha}$.
- **Threshold query**, i.e. a conjunctive query that, given an integer s and an operator $op \in \{=, <, >, \leq, \geq\}$, outputs all the answers having their inconsistency degree d w.r.t. α satisfying the condition: $d\ op\ s$. In the sequel, a threshold query is denoted $Q^{s,op,\alpha}$. The system INCA enables to show the maximum and the minimum inconsistent degrees that the set of answers of a query can have depending to the set of measures of inconsistency degrees selected.

The GUI of the Query I-Profiling module is depicted in Figure 8.5. As input, the user provides the following parameters:

- the subset of constraints to be considered, the format of a denial constraint is as follows: constraint is composed from two parts and the parts are separated by " : ", the first part is the list of relations in the denial constraint (each one is followed by an alias) and the second part is a conjunction of built-in atoms from predicates $\{\leq, \geq, <, >, =, \neq\}$ and the set of attributes of relations in the first part and constants (example: $Person\ t_1$, $Person\ t_2$: $t_1.id = t_2.id\ AND\ t_1.name \neq t_2.name$ that is the denial constraint $\neg(Person(id, name1, \ldots), Person(id, name2, \ldots), name1 \neq name2)$: with one person is associated only one name);
- the input query, that is a conjunctive query in SQL format and each relation in join is associated to an alias;
- the type of query (i.e., top-K query vs. threshold query) with its associated parameters; and
- the inconsistency measure α to consider ($\alpha \in \{CBM, CBS, CSM_{min}, CSS_{min}\}$).

For both classes of queries, the Query I-Profiling module returns the output of the query, together with the inconsistency degree associated with each answer. Additional statistical information is computed and displayed as show in Figure 8.5: distribution of answers by inconsistency level and by subset of violated constraints.

To assist users in specifying the parameter s of a threshold query $Q^{s,op,\alpha}$, INCA shows the range of possible values for s by computing the *min* and *max* inconsistency degrees of the answers of the associated conjunctive query Q. Interestingly, INCA uses the TopINC algorithm to efficiently compute the *min* and *max* bounds without computing the whole output of the associated conjunctive query Q.

The Query I-Profiling module enables also to analyze query outputs w.r.t. multiple inconsistency measures at a time. In this case, the module provides a synthetic view to summarize the collected information as illustrated in Figure 8.5 for example showing the combined analysis of CBM and CBS.

8.4 Conclusion

In this chapter, we presented the system INCA that contains a set of tools enable to characterize inconsistency. Profiling of inconsistency in whole database and query answers are done by INCA. After extensively described the system INCA, we performed a demonstration using a real dataset (food inspection) with meaningful set of denial constraints.

Chapter 9

CONCLUSION AND PERSPECTIVES

In this thesis, we have addressed the problem of inconsistency in relational databases. For this purpose, we defined a set of measures to quantify inconsistency levels of base tuple and we investigated how to propagate them to query answers. We have grounded the computation of inconsistency degrees in why-provenance annotations and polynomial provenance annotations. In addition, we have designed novel top-k algorithms suitable for the above measures while proving their optimality. These proposed algorithms are derived from a base algorithm, called TopINC. We designed experiments to evaluate empirically the performance of the TopINCalgorithm. We developed a tool (called INCA) that can be used for data profiling in the context of inconsistency database.

In the following, we give some perspectives that can be derived following the work conducted in this thesis.

- It would be interesting to extend our work to more expressive query languages as for example negation and aggregation. For example, in the case of the presence of negation in the query the connection between provenance and inconsistency degrees of base tuples remains unclear and raises intriguing research questions.
- Explore more inconsistency handling in the case where the constraints are inconsistent. One interesting research problem would be to design an efficient algorithm to handle the top-k problem for the measure R^{f}_{CBM}. It would also be interesting to enhance the approach proposed in this thesis so that the $orderVio(DC, f)$ list is eliminated or not entirely materialized. Finally, we envision to explore additional measures of inconsistency degrees, taking account the inconsistency of constraints, in the case of set semantic of query answers.
- It remains open whether it is possible to design efficient top-k algorithms, that go behind the naive algorithm for the following measures of inconsistency degrees
  - TBS when the query is a self-join conjunctive query
  - TSS_{min}, TSS_{max}, TSM_{min}, TSM_{max}, CSM_{min}, CSM_{max}, CSS_{min}, CSS_{max} without any particular condition on these measures
- From the practical point of view, it would be useful to conduct a comparative study of measures of inconsistency degrees to identify in what application context a given measure is more adequate.

- Another research direction would be to explore the properties of the proposed measures, in the spirit of [109], as for example the **Reliability** that ensures that any measure of inconsistency gives an inconsistency value greater than zero when any tuple in the database is inconsistent.

Bibliography

[1] `https://github.com/HPI-Information-Systems/Metanome`.

[2] On inference from inconsistent premisses. *Theory and Decision*, pages 179–217, 1970.

[3] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Data profiling: A tutorial. In *Proceedings of ACM SIGMOD*, pages 1747–1751, 2017.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] Sihem Amer-Yahia, Shady Elbassuoni, Ahmad Ghizzawi, Ria Mae Borromeo, Emilie Hoareau, and Philippe Mulhem. Fairness in online jobs: A case study on taskrabbit and google. In *EDBT 2020*, pages 510–521, 2020.

[6] Aziz Amezian El Khalfioui, Jonathan Joertz, Dorian Labeeuw, Gaëtan Staquet, and Jef Wijsen. Optimization of answer set programs for consistent query answering by means of first-order rewriting. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 25–34. Association for Computing Machinery.

[7] Meriem Ammoura, Raddaoui Badran, Yakoub Salhi, and Brahim Oukacha. *On Measuring Inconsistency Using Maximal Consistent Sets*. July 2015. Pages: 276.

[8] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Scalar aggregation in FD-inconsistent databases. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory — ICDT 2001*, Lecture Notes in Computer Science, pages 39–53. Springer.

[9] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *ACM PODS 1999*, pages 68–79, 1999.

[10] Ofer Arieli and Arnon Avron. The value of the four values. *Artificial Intelligence*, 102(1):97–141, June 1998.

[11] Abdallah Arioua and Angela Bonifati. User-guided repairing of inconsistent knowledge bases. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 133–144, 2018.

[12] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *In FOCS, IEEE*, page 739–748, 2008.

[13] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Proceedings of the 21st International Conference, and Proceedings of the 16th Annuall Conference on Computer Science Logic*, CSL'07/EACSL'07, pages 208–222, Berlin, Heidelberg, 2007. Springer-Verlag. event-place: Lausanne, Switzerland.

[14] Carlo Batini, Cinzia Cappiello, Chiara Francalanci, and Andrea Maurino. Methodologies for data quality assessment and improvement. *ACM Comput. Surv.*, 41(3), July 2009.

[15] Carlo Batini and Monica Scannapieco. *Data Quality Issues in Data Integration Systems*, pages 279–307. 03 2016.

[16] Marianne Baudinet, Jan Chomicki, and Pierre Wolper. Constraint-Generating Dependencies. *Journal of Computer and System Sciences*, 59(1):94–115, 1999.

[17] Nuel D. Belnap. A Useful Four-Valued Logic. In J. Michael Dunn and George Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 5–37. Springer Netherlands, Dordrecht, 1977.

[18] Salem Benferhat, Didier Dubois, and Henri Prade. Some Syntactic Approaches to the Handling of Inconsistent Knowledge Bases: A Comparative Study Part 1: The Flat Case. *Studia Logica*, 58(1):17–45, January 1997.

[19] L. Bertossi, L. Bravo, E. Franconi, and A. Lopatenko. Fixing inconsistent databases by updating numerical attributes. In *16th International Workshop on Database and Expert Systems Applications (DEXA'05)*, pages 854–858. ISSN: 2378-3915.

[20] Leopoldo Bertossi. Consistent query answering in databases. 35(2):68–76.

[21] Leopoldo Bertossi. Database repairs and consistent query answering: Origins and further developments. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '19, pages 48–58. Association for Computing Machinery.

[22] Leopoldo Bertossi. *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers, 2011.

[23] Leopoldo Bertossi and Jan Chomicki. *Query Answering in Inconsistent Databases*, pages 43–83. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[24] Leopoldo Bertossi, Anthony Hunter, and Torsten Schaub. Inconsistency Tolerance [result from a Dagstuhl seminar]. January 2005.

[25] Philippe Besnard and Anthony Hunter. Quasi-classical logic: Non-trivializable classical reasoning from inconsistent information. In Christine Froidevaux and Jürg Kohlas, editors, *Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 44–51, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[26] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2):153–187, June 2002.

[27] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In *Database Theory — ICDT*, pages 316–330.

[28] W. A. Carnielli and J. Marcos. A Taxonomy of C-systems. *arXiv:math/0108036*, August 2001. arXiv: math/0108036.

[29] Kevin Chen-Chuan Chang and Seung-won Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, page 346–357, New York, NY, USA, 2002. Association for Computing Machinery.

[30] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R. Smith. The onion technique: Indexing for linear optimization queries. *SIGMOD Rec.*, 29(2):391–402, May 2000.

[31] Jan Chomicki. Consistent query answering: The first ten years. In Sergio Greco and Thomas Lukasiewicz, editors, *Scalable Uncertainty Management*, Lecture Notes in Computer Science, pages 1–3. Springer.

[32] Jan Chomicki and Jerzy Marcinkowski. Minimal-change integrity maintenance using tuple deletions. 197(1):90–121. Place: USA Publisher: Academic Press, Inc.

[33] Jan Chomicki and Jerzy Marcinkowski. On the computational complexity of minimal-change integrity maintenance in relational databases. In Leopoldo Bertossi, Anthony Hunter, and Torsten Schaub, editors, *Inconsistency Tolerance*, Lecture Notes in Computer Science, pages 119–150. Springer.

[34] Jan Chomicki, Jerzy Marcinkowski, and Slawomir Staworko. Computing consistent query answers using conflict hypergraphs. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, CIKM '04, pages 417–426. Association for Computing Machinery.

[35] Jan Chomicki, Jerzy Marcinkowski, and Slawomir Staworko. Hippo: A system for computing consistent answers to a class of SQL queries. In Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors, *Advances in Database Technology - EDBT 2004*, Lecture Notes in Computer Science, pages 841–844. Springer.

[36] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 2201–2206, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Discovering denial constraints. *Proc. VLDB Endow.*, 6(13):1498–1509, August 2013.

[38] Newton C. A. da Costa. On the theory of inconsistent formal systems. *Notre Dame Journal of Formal Logic*, 15(4):497–510, October 1974. Publisher: Duke University Press.

[39] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. Answering top-k queries using views. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 451–462. VLDB Endowment, 2006.

[40] Glauber De Bona, John Grant, Anthony Hunter, and Sébastien Konieczny. Classifying Inconsistency Measures Using Graphs. *Journal of Artificial Intelligence Research*, 2019.

[41] Hendrik Decker. *Quantifying the Quality of Stored Data by Measuring their Integrity*. September 2009. Pages: 801.

[42] Hendrik Decker. *Causes of the Violation of Integrity Constraints for Supporting the Quality of Databases*, volume 6786. June 2011. Pages: 292.

[43] Hendrik Decker. *Inconsistency-Tolerant Integrity Checking Based on Inconsistency Metrics*. September 2011. Pages: 558.

[44] Hendrik Decker and Davide Martinenghi. *A Relaxed Approach to Integrity and Inconsistency in Databases*. November 2006. Pages: 301.

[45] Hendrik Decker and Davide Martinenghi. *Classifying integrity checking methods with regard to inconsistency tolerance*. January 2008. Pages: 204.

[46] Hendrik Decker and Davide Martinenghi. *Modeling, Measuring and Monitoring the Quality of Information*, volume 5833. November 2009. Pages: 221.

[47] Hendrik Decker, Francesc Muñoz-Escoí, and Sanjay Misra. *Data Consistency: Toward a Terminological Clarification*. August 2015.

[48] Shaleen Deep and Paraschos Koutris. Ranked enumeration of conjunctive query results.

[49] Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. Circuits for datalog provenance. In *ICDT 2014*, 03 2014.

[50] Dragan Doder, Miodrag Rašković, Zoran Marković, and Zoran Ognjanović. Measures of inconsistency and defaults. *International Journal of Approximate Reasoning*, 51:832–845, September 2010.

[51] Ronald Fagin. Combining Fuzzy Information from Multiple Systems. *Journal of Computer and System Sciences*, 58(1):83–99, February 1999.

[52] Ronald Fagin, Benny Kimelfeld, and Phokion G. Kolaitis. Dichotomies in the complexity of preferred repairs. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '15, pages 3–15. Association for Computing Machinery.

[53] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, page 102–113, New York, NY, USA, 2001. Association for Computing Machinery.

[54] Wenfei Fan. Data quality: From theory to practice. *SIGMOD Rec.*, 44(3):7–18, December 2015.

[55] Wenfei Fan and Floris Geerts. Foundations of Data Quality Management. *Synthesis Lectures on Data Management*, 4(5):1–217, July 2012. Publisher: Morgan & Claypool Publishers.

[56] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2), June 2008.

[57] Craig W. Fisher and Bruce R. Kingma. Criticality of data quality as exemplified in two disasters. 39(2):109–116, 2001.

[58] Sergio Flesca, Filippo Furfaro, and Francesco Parisi. Consistent answers to boolean aggregate queries under aggregate constraints. In Pablo García Bringas, Abdelkader Hameurlain, and Gerald Quirchmayr, editors, *Database and Expert Systems Applications*, Lecture Notes in Computer Science, pages 285–299. Springer.

[59] Sergio Flesca, Filippo Furfaro, and Francesco Parisi. Range-consistent answers of aggregate queries under aggregate constraints. In Amol Deshpande and Anthony Hunter, editors, *Scalable Uncertainty Management*, Lecture Notes in Computer Science, pages 163–176. Springer.

[60] Gaëlle Fontaine. Why is it hard to obtain a dichotomy for consistent query answering? In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 550–559. ISSN: 1043-6871.

[61] Ariel Fuxman and Renée J. Miller. First-order query rewriting for inconsistent databases. 73(4):610–635.

[62] Ariel D. Fuxman and Renée J. Miller. First-order query rewriting for inconsistent databases. In Thomas Eiter and Leonid Libkin, editors, *Database Theory - ICDT 2005*, Lecture Notes in Computer Science, pages 337–351. Springer.

[63] Jaffer Gardezi and Leopoldo Bertossi. Query rewriting using datalog for duplicate resolution. In Pablo Barceló and Reinhard Pichler, editors, *Datalog in Academia and Industry*, Lecture Notes in Computer Science, pages 86–98. Springer.

[64] Floris Geerts, Fabian Pijcke, and Jef Wijsen. First-order under-approximations of consistent query answers. 83.

[65] Nathan Goodman, Oded Shmueli, and Y. C. Tay. GYO reductions, canonical connections, tree and cyclic schemas, and tree projections. *Journal of Computer and System Sciences*, 29(3):338 – 358, 1984.

[66] Georg Gottlob and Marko Samer. A Backtracking-Based Algorithm for Hypertree Decomposition. *ACM J. Exp. Algorithmics*, 13, February 2009. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[67] John Grant. Classifications for inconsistent theories. *Notre Dame Journal of Formal Logic*, 19(3):435–444, July 1978. Publisher: Duke University Press.

[68] John Grant and Anthony Hunter. Measuring inconsistency in knowledgebases. *Journal of Intelligent Information Systems*, 27(2):159–184, September 2006.

[69] John Grant and Anthony Hunter. Analysing inconsistent first-order knowledgebases. *Artificial Intelligence*, 172:1064–1093, May 2008.

[70] John Grant and Francesco Parisi. General information spaces: measuring inconsistency, rationality postulates, and complexity. *Annals of Mathematics and Artificial Intelligence*, pages 1–35, April 2021.

[71] John Grant and V. S. Subrahmanian. Applications Of Paraconsistency In Data And Knowledge Bases. *Synthese*, 125(1):121–132, January 2000.

[72] Sergio Greco and Cristian Molinaro. Probabilistic query answering over inconsistent databases. 64(2):185–207.

[73] Todd J. Green. Containment of conjunctive queries on annotated relations. In *ICDT '09 Proceedings of the 12th International Conference on Database Theory*, pages 296–309. ACM, 2009.

[74] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *ACM PODS 2007*, pages 31–40. ACM, 2007.

[75] Luca Grieco, Domenico Lembo, Riccardo Rosati, and Marco Ruzzi. Consistent query answering under key and exclusion dependencies: algorithms and experiments. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, CIKM '05, pages 792–799. Association for Computing Machinery.

[76] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, page 419–428, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[77] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, page 287–298, New York, NY, USA, 1999. Association for Computing Machinery.

[78] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. *SIGMOD Rec.*, 30(2):259–270, May 2001.

[79] A. Hunter and B. Nuseibeh. Managing inconsistent specifications: reasoning, analysis, and action. *ACM Trans. Softw. Eng. Methodol.*, 7:335–367, 1998.

[80] Anthony Hunter. Paraconsistent Logics. In Philippe Besnard and Anthony Hunter, editors, *Reasoning with Actual and Potential Contradictions*, pages 11–36. Springer Netherlands, Dordrecht, 1998.

[81] Anthony Hunter, Glauber De Bona, John Grant, and Sébastien Konieczny. Towards a Unified Framework for Syntactic Inconsistency Measures. February 2018.

[82] Anthony Hunter and Sébastien Konieczny. *Approaches to Measuring Inconsistent Information*, volume 3300. January 2005. Pages: 236.

[83] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Joining ranked inputs in practice. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, page 950–961. VLDB Endowment, 2002.

[84] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, page 754–765. VLDB Endowment, 2003.

[85] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-*k* query processing techniques in relational database systems. 40(4). Place: New York, NY, USA Publisher: Association for Computing Machinery.

[86] Ousmane Issa, Angela Bonifati, and Farouk Toumani. A relational framework for inconsistency-aware query answering. In *BDA, conference des bases de données avancées*, October 2019.

[87] Ousmane Issa, Angela Bonifati, and Farouk Toumani. Evaluating top-k queries with inconsistency degrees. 2020.

[88] Said Jabbour. On inconsistency measuring and resolving. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, ECAI'16, pages 1676–1677, NLD, August 2016. IOS Press.

[89] Said Jabbour and Raddaoui Badran. *Measuring Inconsistency Through Minimal Proofs*, volume 7958. July 2013. Journal Abbreviation: LNCS Pages: 301 Publication Title: LNCS.

[90] Said Jabbour, Raddaoui Badran, and Lakhdar Sais. *Knowledge Base Compilation for Inconsistency Measures*. January 2016. Pages: 539.

[91] Said Jabbour, Yue Ma, Raddaoui Badran, and Lakhdar Sais. Quantifying Conflicts in Propositional Logic Through Prime Implicates. *International Journal of Approximate Reasoning*, 89, January 2017.

[92] Phokion G. Kolaitis and Enela Pema. A dichotomy in the complexity of consistent query answering for queries with two atoms. 112(3):77–85.

[93] Phokion G. Kolaitis, Enela Pema, and Wang-Chiew Tan. Efficient querying of inconsistent databases with binary integer programming. 6(6):397–408.

[94] Sebastien Konieczny, Jerome Lang, and Pierre Marquis. Quantifying Information and Contradiction in Propositional Logic through Test Actions. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 106–111, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc. event-place: Acapulco, Mexico.

[95] Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys. 45(1):15–22.

[96] Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys and conjunctive queries with negated atoms. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, SIGMOD/PODS '18, pages 209–224. Association for Computing Machinery.

[97] Paraschos Koutris and Jef Wijsen. Consistent query answering for self-join-free conjunctive queries under primary key constraints. 42(2).

[98] Paraschos Koutris and Jef Wijsen. First-order rewritability in consistent query answering with respect to multiple keys. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS'20, pages 113–129. Association for Computing Machinery.

[99] Nuno Laranjeiro, Seyma Nur Soydemir, and Jorge Bernardino. A survey on data quality: Classifying poor data. In *2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 179–188, 2015.

[100] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, page 233–246, New York, NY, USA, 2002. Association for Computing Machinery.

[101] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. Ranksql: Query algebra and optimization for relational top-k queries. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 131–142, New York, NY, USA, 2005. Association for Computing Machinery.

[102] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Supporting ad-hoc ranking aggregates. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 61–72. Association for Computing Machinery. event-place: Chicago, IL, USA.

[103] Jie Liu, Fei Huang, Dan Ye, and Tao Huang. Efficient consistent query answering based on attribute deletions. In *International Symposium on Computer Science and its Applications*, pages 222–227. ISSN: 2159-7049.

[104] Jie Liu, Dan Ye, Jun Wei, Fei Huang, and Hua Zhong. Consistent query answering based on repairing inconsistent attributes with nulls. In Weiyi Meng, Ling Feng, Stéphane Bressan, Werner Winiwarter, and Wei Song, editors, *Database Systems for Advanced Applications*, Lecture Notes in Computer Science, pages 407–423. Springer.

[105] Ester Livshits and Benny Kimelfeld. Counting and enumerating (preferred) database repairs. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17, pages 289–301. Association for Computing Machinery.

[106] Ester Livshits, Benny Kimelfeld, and Sudeepa Roy. Computing optimal repairs for functional dependencies. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, SIGMOD/PODS '18, pages 225–237. Association for Computing Machinery.

[107] Ester Livshits, Benny Kimelfeld, and Sudeepa Roy. Computing optimal repairs for functional dependencies. 45(1):4:1–4:46.

[108] Ester Livshits, Benny Kimelfeld, and Jef Wijsen. Counting subset repairs with functional dependencies. 117:154–164.

[109] Ester Livshits, Rina Kochirgan, Segev Tsur, Ihab Ilyas, Benny Kimelfeld, and Sudeepa Roy. *Properties of Inconsistency Measures for Databases*. June 2021. Pages: 1194.

[110] A. Lopatenko and L. Bertossi. Consistent query answering by minimal-size repairs. In *17th International Workshop on Database and Expert Systems Applications (DEXA'06)*, pages 558–562. ISSN: 2378-3915.

[111] Andrei Lopatenko and Loreto Bravo. Efficient approximation algorithms for repairing inconsistent databases. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 216–225. ISSN: 2375-026X.

[112] Yue Ma, Guilin Qi, and Pascal Hitzler. Computing Inconsistency Measure based on Paraconsistent Semantics. *Journal of Logic and Computation*, 21, December 2011.

[113] Yue Ma, Guilin Qi, Pascal Hitzler, and Zuoquan Lin. *Measuring Inconsistency for Description Logics Based on Paraconsistent Semantics*, volume 250. September 2007. Journal Abbreviation: CEUR Workshop Proceedings Pages: 41 Publication Title: CEUR Workshop Proceedings.

[114] Nicolás Madrid and Manuel Ojeda-Aciego. Measuring Inconsistency in Fuzzy Answer Set Semantics. *Fuzzy Systems, IEEE Transactions on*, 19:605–622, September 2011.

[115] Amélie Marian, Nicolas Bruno, and Luis Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, June 2004.

[116] Maria Vanina Martinez, Andrea Pugliese, Gerardo Simari, V. Subrahmanian, and Henri Prade. *How Dirty Is Your Relational Database? An Axiomatic Approach*. October 2007. Pages: 114.

[117] Dany Maslowski and Jef Wijsen. On counting database repairs. In *Proceedings of the 4th International Workshop on Logic in Databases*, LID '11, pages 15–22. Association for Computing Machinery.

[118] Cristian Molinaro. Polynomial time queries over inconsistent databases. In Sergio Greco and Thomas Lukasiewicz, editors, *Scalable Uncertainty Management*, Lecture Notes in Computer Science, pages 312–325. Springer.

[119] Cristian Molinaro, Jan Chomicki, and Jerzy Marcinkowski. Disjunctive databases for representing repairs. 57(2):103–124.

[120] Cristian Molinaro and Sergio Greco. Polynomial time queries over inconsistent databases with functional dependencies and foreign keys. 69(7):709–722.

[121] Kedian Mu, Weiru Liu, and Zhi Jin. A general framework for measuring inconsistency through minimal inconsistent sets. *Knowl. Inf. Syst.*, 27:85–114, April 2011.

[122] Kedian Mu, Kewen Wang, and Lian Wen. Approaches to measuring inconsistency for stratified knowledge bases. *International Journal of Approximate Reasoning*, 55, January 2013.

[123] David Muino. Measuring and repairing inconsistency in knowledge bases with graded truth. *Fuzzy Sets and Systems - FSS*, 197, January 2011.

[124] David Picado Muiño. Measuring and repairing inconsistency in probabilistic knowledge bases. *International Journal of Approximate Reasoning*, 52(6):828–840, 2011.

[125] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, page 281–290, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[126] Eduardo H. M. Pena, Eduardo C. de Almeida, and Felix Naumann. Discovery of approximate (and exact) denial constraints. *Proc. VLDB Endow.*, 13(3):266–278, November 2019.

[127] Thomas C. Redman and A. Blanton Godfrey. *Data Quality for the Information Age*. Artech House, Inc., USA, 1st edition, 1997.

[128] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[129] Yakoub Salhi. Measuring Inconsistency Through Subformula Forgetting. pages 184–191. December 2019.

[130] Karl Schnaitter and Neoklis Polyzotis. Evaluating rank joins with optimal cost. PODS '08, page 43–52, New York, NY, USA, 2008. Association for Computing Machinery.

[131] Karl Schnaitter and Neoklis Polyzotis. Optimal algorithms for evaluating rank joins in database systems. *ACM Trans. Database Syst.*, 35(1), February 2008.

[132] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. Provsql: Provenance and probability management in postgresql. *Proc. VLDB Endow.*, 11(12):2034–2037, August 2018.

[133] Fatimah Sidi, Payam Hassany Shariat Panahy, Lilly Suriani Affendey, Marzanah A. Jabar, Hamidah Ibrahim, and Aida Mustapha. Data quality: A survey of data quality dimensions. In *2012 International Conference on Information Retrieval Knowledge Management*, pages 300–304, 2012.

[134] Matthias Thimm. Measuring Inconsistency in Probabilistic Knowledge Bases. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, pages 530–537, Arlington, Virginia, USA, 2009. AUAI Press. event-place: Montreal, Quebec, Canada.

[135] Matthias Thimm. *Towards Large-Scale Inconsistency Measurement*, volume 8736. May 2015.

[136] Matthias Thimm. On the Compliance of Rationality Postulates for Inconsistency Measures: A More or Less Complete Picture. *KI - Künstliche Intelligenz*, August 2016.

[137] Matthias Thimm. *Uncertainty and Inconsistency in Knowledge Representation (Habilitation Thesis)*. February 2016.

[138] Matthias Thimm. Measuring Inconsistency with Many-Valued Logics. *International Journal of Approximate Reasoning*, 86, April 2017.

[139] Trong Hieu Tran and Ngoc Thanh Nguyen. Integration of Knowledge in Disjunctive Structure on Semantic Level. In Ignac Lovrek, Robert J. Howlett, and Lakhmi C. Jain, editors, *Knowledge-Based Intelligent Information and Engineering Systems*, Lecture Notes in Computer Science, pages 253–261, Berlin, Heidelberg, 2008. Springer.

[140] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Optimal join algorithms meet top-k. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2659–2665, New York, NY, USA, 2020. Association for Computing Machinery.

[141] Markus Ulbricht. *Understanding Inconsistency – A Contribution to the Field of Non-monotonic Reasoning*. PhD thesis, October 2019.

[142] Markus Ulbricht, Matthias Thimm, and Gerhard Brewka. *Measuring Strong Inconsistency*. November 2017.

[143] Markus Ulbricht, Matthias Thimm, and Gerhard Brewka. Inconsistency Measures for Disjunctive Logic Programs Under Answer Set Semantics. February 2018.

[144] Markus Ulbricht, Matthias Thimm, and Gerhard Brewka. Handling and Measuring Inconsistency in Non-monotonic Logics. *Artificial Intelligence*, 286:103344, June 2020.

[145] Richard Y. Wang and Diane M. Strong. Beyond accuracy: What data quality means to data consumers. *J. Manage. Inf. Syst.*, 12(4):5–33, March 1996.

[146] J. Wijsen. On condensing database repairs obtained by tuple deletions. In *16th International Workshop on Database and Expert Systems Applications (DEXA'05)*, pages 849–853. ISSN: 2378-3915.

[147] Jef Wijsen. Certain conjunctive query answering in first-order logic. 37(2):9:1–9:35.

[148] Jef Wijsen. Condensed representation of database repairs for consistent query answering. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory — ICDT 2003*, Lecture Notes in Computer Science, pages 378–393. Springer.

[149] Jef Wijsen. Consistent query answering under primary keys: a characterization of tractable queries. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 42–52. Association for Computing Machinery.

[150] Jef Wijsen. Project-join-repair: An approach to consistent query answering under functional dependencies. In Henrik Legind Larsen, Gabriella Pasi, Daniel Ortiz-Arroyo, Troels Andreasen, and Henning Christiansen, editors, *Flexible Query Answering Systems*, Lecture Notes in Computer Science, pages 1–12. Springer.

[151] Jef Wijsen. A survey of the data complexity of consistent query answering under key constraints. In Christoph Beierle and Carlo Meghini, editors, *Foundations of Information and Knowledge Systems*, Lecture Notes in Computer Science, pages 62–78. Springer International Publishing.

[152] Jef Wijsen. Database repairing using updates. *ACM Trans. Database Syst.*, 30(3):722–768, September 2005.

[153] Guohui Xiao and Yue Ma. Inconsistency Measurement based on Variables in Minimal Unsatisfiable Subsets. *Frontiers in Artificial Intelligence and Applications*, 242, August 2012.

[154] Zhang Xiaowang and Zuoquan Lin. Quasi-Classical Description Logic. *Journal of Multiple-Valued Logic and Soft Computing*, 18, January 2012.

[155] Dong Xie, Xinbo Chen, and Yan Zhu. Obtaining certain results by query rewriting over uncertain database. In Wei Zhang, editor, *Software Engineering and Knowledge Engineering: Theory and Practice*, Advances in Intelligent and Soft Computing, pages 401–405. Springer.

[156] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, page 82–94. VLDB Endowment, 1981.

[157] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 82–94. VLDB Endowment, 1981. event-place: Cannes, France.

[158] C. Yu and M. Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *COMPSAC 79 - Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference*, pages 306,307,308,309,310,311,312, Los Alamitos, CA, USA, nov 1979. IEEE Computer Society.

[159] Du Zhang. *Quantifying Knowledge Base Inconsistency Via Fixpoint Semantics*. September 2007. Journal Abbreviation: Proceedings of the 6th IEEE International Conference on Cognitive Informatics, ICCI 2007 Pages: 262 Publication Title: Proceedings of the 6th IEEE International Conference on Cognitive Informatics, ICCI 2007.

[160] Zhen Zhang, Seung-won Hwang, Kevin Chen-Chuan Chang, Min Wang, Christian A. Lang, and Yuan-chi Chang. Boolean + ranking: Querying a database by k-constrained optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 359–370, New York, NY, USA, 2006. Association for Computing Machinery.