Evaluating Top-k Queries with Inconsistency Degrees

Ousmane Issa
University Clermont Auvergne
ousmane.issa@uca.fr

Angela Bonifati
Lyon 1 University
angela.bonifati@univ-lyon1.fr

Farouk Toumani
University Clermont Auvergne
farouk.toumani@uca.fr

ABSTRACT

We study the problem of augmenting relational tuples with inconsistency awareness and tackling top-k queries under a set of denial constraints (DCs). We define a notion of inconsistent tuples with respect to a set of DCs and define two measures of inconsistency degrees, which consider single and multiple violations of constraints. In order to compute these measures, we leverage two models of provenance, namely why-provenance and provenance polynomials. We investigate top-k queries that allow to rank the answer tuples by their inconsistency degrees. Since one of our measure is monotonic and the other non-monotonic, we design an integrated top-k algorithm to compute the top-k results of a query w.r.t. both inconsistency measures. By means of an extensive experimental study, we gauge the effectiveness of inconsistency-aware query answering and the efficiency of our algorithm with respect to a baseline, where query results are fully computed and ranked afterwards.

PVLDB Reference Format: Ousmane Issa, Angela Bonifati, Farouk Toumani. Evaluating Top-k Queries with Inconsistency Degrees.. *PVLDB*, 13(11): 2146-2158, 2020. DOI: https://doi.org/10.14778/3407790.3407815

1. INTRODUCTION

Assessing the quality of raw data is crucial in numerous data management tasks, spanning from traditional querying and data integration to cutting-edge in-database analytical and inference/ML processes. Throughout these processes, the quality of the output as well as the trustworthiness of the results might be tremendously affected by the quality of the input tuples [29].

Past work on data curation for relational tuples has mainly addressed the problem of detecting and repairing the violations with respect to a set of constraints [40]. Consistent query answering has also been considered as a mean to reconcile several possible repairs of the original data [11]. However, little or no attention has been paid to leave the database instances intact and quantifying their degrees of inconsistency at different levels of granularity (tuple, sets of tuples, entire relations) as we do in this paper for the first time. 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. To this end, we focus on top-k rank join queries over an inconsistent relational database in the presence of a set of Denial Constraints (DCs) [17]. The inconsistency degree of an answer tuple of a given query is determined by relying on provenance-based information of the input tuples. We first leverage why-provenance 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 [26] in order to propagate the annotations of inconsistencies from the base tuples to the answer tuples of Conjunctive Queries (CQs). Building upon the computed annotations, we define two measures of inconsistency degrees, which consider single and multiple violations of constraints. Since one of our measures is a monotone function and the other a non-monotone function, we design an integrated top-k algorithm to rank the top-k results of a query w.r.t. the above inconsistency measures. 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 [38]. 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 [3].

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407815

In this paper, we make the following main contributions: **(1)** We design novel measures of inconsistency degrees of answer tuples for CQs over an inconsistent database in the presence of a set of DCs.
(2) We leverage why-provenance in order to identify the inconsistent tuples of a given database w.r.t. a set of DCs. Then, we exploit provenance polynomials to propagate the inconsistency degree of base tuples to the answer tuples of CQs, which is to the best of our knowledge a novel and quite promising usage of provenance. Both provenance techniques considered here are PTIME-computable in data complexity thus guaranteeing that our augmentation of relational instances with inconsistency measures remains tractable.
(3) We study the problem of top-k rank queries with respect to a given measure of inconsistency degree. We consider monotonic and non-monotonic measures and we design an integrated top-k ranking algorithm that is applicable to both. We show that the algorithm is space efficient (i.e., in $O(|\mathcal{I}| + k)$ instead of $O(Q(\mathcal{I}))$, where $\mathcal{I}$ is a database instance). As a side contribution, we prove that our algorithm is optimal w.r.t. a new notion of optimality tailored to generic scoring functions (monotonic and non-monotonic).
(4) We deploy our framework with real-world and synthetic datasets equipped with DCs. We gauge the low overhead due to the generation of inconsistency degrees as well as the bearable runtimes of our top-k ranking algorithm with various query sets. Our experimental study shows the feasibility of inconsistency-aware query answering with top-k ranking, with the latter outperforming by up to 2^8 times the baseline solution.

The paper is organized as follows: Section 2 introduces a motivating example; Section 3 and Section 4 present the state of the art and the background notions, respectively; Section 5 introduces the inconsistency measures; Section 6 describes our novel top-k algorithm along with its properties. Section 7 illustrates our experimental study while Section 8 concludes the paper and pinpoints future directions.

2. MOTIVATING EXAMPLE

Consider a relational instance $\mathcal{I}$ as depicted in Figure 1 consisting of three relations D, V and S contaning information about the diagnoses, vaccinations and surgical interventions of patients. Hence, in each of the relations D, V and S, the first column represents the patient identifier *PID*, the second column is the disease identifier *RefID* and the third column is the *Date* of a given event [1].

The denial constraint C_1 imposes to have any diagnosis for a patient's disease before surgery for the same disease concerning that patient. The constraint C_2 establishes that a patient cannot be diagnosed a given disease for which he/she has been administered a vaccine on a previous date. The constraint C_3 requires that a patient cannot undergo surgery and vaccination on the same date. Figure 1 also shows a conjunctive query Q_{ex} extracting pairs of diseases for which the same patient underwent surgery and was administered a vaccine.

Observe that the instance $\mathcal{I}$ does not satisfy the set of denial constraint $\{C_1, C_2, C_3\}$. Indeed, the constraint C_1 is violated both by the pairs of tuples (t_2, t_3) and (t_2, t_4) while the constraint C_2 is violated by the tuples (t_2, t_7) and the constraint C_3 is violated by the tuples (t_4, t_7). Hence, the inconsistent base tuples of $\mathcal{I}$ w.r.t. the set of denial constraint $\{C_1, C_2, C_3\}$ are: t_2, t_3, t_4, t_7.

Quantifying the inconsistency degrees of query answers. Evaluating the query Q_{ex} over the inconsistent database $\mathcal{I}$, under the bag answer semantics, returns the answers reported in the first column of Table 1. The second column of Table 1 shows the base tuples of $\mathcal{I}$ that contribute to each answer. One can observe that the tuple $\langle d4, d4\rangle$ is a consistent answer since it is computed using consistent base tuples. This is, however, not the case for $\langle d2, d2\rangle$, which can be derived in three possible ways, either using the base tuples $\{t_2,\ t_3,\ t_7\}$ or $\{t_2,\ t_4,\ t_7\}$ or $\{t_2,\ t_6,\ t_7\}$. Notice that all the above derivations use inconsistent base tuples.

In this paper, we consider two alternatives for quantifying the inconsistency degree of query answers. The first approach quantifies the inconsistency degree of an answer t by counting the number of constraints violated by the base tuples that contribute to t. The third column of Table 1 shows the constraints violated by the base tuples that contribute to each answer. For example, in line 1 of Table 1, the computation of the answer $\langle d2, d2\rangle$ from the base tuples $\{t_2,\ t_3,\ t_7\}$ leads to the violation of the constraints C_1 (violated by t_2 and t_3), C_2 (violated by t_2 and t_7) and C_3 (violated by t_7). As a consequence, the inconsistency degree of the answer $\langle d2, d2\rangle$ when it is computed from the base tuples $\{t_2,\ t_3,\ t_7\}$ is equal to 3 (i.e., 3 constraints are violated by the base tuples that contribute to this answer).

A second approach to quantify the inconsistency degree of an answer amounts to counting the number of violations of the constraints per each base tuple. The fourth column of Table 1 shows the constraints violated by the base tuples while reporting how many times the same violation occurred (indicated by the exponent of the constraints). For instance, in the first row of Table 1, the computation of the answer $\langle d2, d2\rangle$ from the base tuples $\{t_2,\ t_3,\ t_7\}$ leads to the violation of the constraint C_1 twice (violated by t_2 and by t_3), the violation of C_2 two times (violated by t_2 and t_7) and the violation of C_3 one time (violated by t_7). Hence, the corresponding inconsistency degree of $\langle d2, d2\rangle$ is equal to 5 (i.e., 5 violations of the constraints by base tuples).

In this paper, we focus on a *constraint-based* approach to quantify inconsistency degrees of query answers under bag semantics. As Table 1 (fourth column) shows, the annotations of the base tuples are the richest and the most informative ones since they consider the violations in terms of the constraints and their occurrences. In particular, we consider two alternative measures: *(i)* counting the number of constraints violated by the base tuples contributing to a given answer (called in the sequel, *single occurrence* quantification), and *(ii)* summing up the exponents of the constraints violated by base tuples contributing to a an answer (called in the sequel, *multiple occurrence* quantification).

Inconsistency-aware query processing. With the inconsistency measures at hand, we are interested in top-k query processing while taking into account the inconsistency degrees of query answers. This problem can be framed in the general context of top-k rank queries where the inconsistency measures are used as scoring functions that can be exploited to rank the answers of queries. In particular, given a query Q, top-k ranking aims at computing the subset of Q's answers with the top k scores, where the scores correspond

[1]For ease of exposition and to avoid clutter, the relations have the same schema. Our approach is applicable to relations with arbitrary schemas as also shown in our experimental study.

PID	RefD	Date	
02	d4	2	t_1
01	d2	4	t_2

Diagnosis(D)

PID	RefD	Date	
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	
01	d2	3	t_7
02	d4	3	t_8

Vaccination(V)

Constraint Id	Denial Constraint
C_1	$\leftarrow\ D(x,y,z) \wedge S(x,y,u) \wedge z > u$
C_2	$\leftarrow\ D(x,y,z) \wedge V(x,y,u) \wedge z > u$
C_3	$\leftarrow\ S(x,y,z) \wedge V(x,v,z)$

Set of denial constraints(DCs)

$Q_{ex}(y,u) \quad \leftarrow \quad D(x,y,z) \wedge S(x,y,z_1) \wedge V(x,u,v)$

Query (Q_{ex})

Figure 1: A hospital database hdb with a set of denial constraints (DCs) and a query Q_{ex}.

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

Table 1: Annotated answers of query Q_{ex}

to the inconsistency degrees of the tuples. Continuing with our example, the top-2 answers of the query Q_{ex} over the database $\mathcal{I}$ are: $\langle d4, d4\rangle$, $\langle d2, d2\rangle$, with respective inconsistency degrees 0 and 3 with single occurrence quantification), or, respectively, inconsistency degrees 0 and 4 with multi occurrence quantification, respectively. In the remainder, we will see that both augmenting the instances with inconsistency measures and enabling top-k queries on them are far from being trivial and need special treatment.

3. RELATED WORK

Consistent Query Answering and Repairs. Central to data quality is data consistency that has been extensively investigated in the past. Starting from the pioneering work of [5], there has been a wealth of research on the problem of *consistent query answering in inconsistent databases* [7, 40, 36, 23, 8]. Most of the existing works make a distinction between consistent and inconsistent answers based on the notion of a *database repair* (see [11, 10] for a survey). A repair is a database instance obtained by applying some *minimal changes* on the initial database instance in order to make it compliant with a given set of integrity constraints. Usually, there exist several possible repairs of a given database and for such a reason, following the *certain answers* semantics, consistent answers to a given query are defined as being the intersection of the query answers on all possible repairs of the initial database instance. The problem of *consistent query answering* has been investigated in several settings depending on: *(i)* the definition of repairs, where a repair is defined in terms of the sets of inserted and/or deleted tuples [11, 10, 34, 13], or updates [40, 32, 6] and *(ii)* the considered integrity constraints and the query language. The latter has been studied for a variety of queries and constraints such as first-order queries and binary universal constraints or single functional dependencies [5], union of conjunctive queries and key and inclusion dependencies [14], first-order queries and denial constraints [16], to mention a few.

Recent approaches study a counting variant of consistent query answering [36], i.e., *counting how many repairs satisfy a given query q*. The work in [36] establishes a dichotomy for the #- CERTAINTY(q) problem, where q is a boolean conjunctive query possibly with self-joins and with single-attribute primary-key (PK) constraints. Our work is agnostic to the repairing semantics and computation of repairs and only leverages constraint violations.

Quantifying Inconsistency in KBs. Reasoning in Knowledge Bases (KBs) in the presence of inconsistency is also widely studied in the AI community, where it has been shown that an arbitrary consequent can be entailed from an inconsistent set of premises (see the survey by Lang et al. [33]). Similarly, the role of inconsistency tolerance in KBs has been introduced in [9]. They are more concerned about the co-existence of consistency and inconsistency as separate axioms when doing reasoning in KBs rather than using inconsistency measures to yield inconsistency-aware query answering, as we do in our work.

Another line of research deals with the definition of inconsistency measures in Knowledge Bases (KBs) [19, 24]. Numerous measures of inconsistency in KBs have been proposed based on various paradigms such as information theory [35], possibility theory [21] or quasi-classical logic [24], to mention a few. The ultimate goal is, however, to provide measures to quantify the inconsistency of a KBs in order, for example, to allow comparison of full KBs based on their inconsistency degrees. Extensions of such an approach to the database field has been recently investigated in [10] where inconsistency measures based on various classes of repair semantics are studied. However, their notion of inconsistency measures (despite the name) relies on restoring the consistency of a database, thus on the class of repairs admitted by the database. As such, it is quite different from our notion of inconsistency.

Top-k Query Processing. Top-k query processing algorithms address the efficient computation of the first top-k answers of selection [22], join [27, 31, 39] and aggregate queries [15]. There exists a vast literature on top-k join query processing, see [28] for a survey. However, a basic assumption of several previous papers on this topic is that the scoring function f used to aggregate the scores of the underlying tuples is a monotonic function [15, 28, 31, 39]. Non-monotonic scoring functions have been considered in [41] for top-k query processing, while assuming that attribute values are indexed by tree-structured indexes (e.g., B-tree, R-tree). However, none of the existing top-k ranking algorithms would be optimal to handle both monotonic and non-monotonic functions. Additionally, our novel top-k algorithm is tailored for inconsistency measures and leverages a suitable cost model for them. Our algorithm is thus generic and designed in a way that it incorporates both monotonic and non-monotonic inconsistency measures.

Analytical studies of the performance of existing top-k join algorithms have been conducted mostly for the class of algorithms, introduced initially in [22], that use a sorted data access strategy, i.e., the input relations are scanned sequentially in base score order [31, 39]. Such a framework, which imposes a sequential data access strategy, is not suitable for non-monotonic functions. This motivated our work on a new notion of optimality based on: *(i)* a more general class of algorithms, called Semi-Blind Algorithms (SBA), that alleviate the assumptions on the permitted data access strategies, and *(ii)* a new cost model that enable to deal

$M(P)$	Set of monomials of a polynomial P
$Var(M)$	Set of variables of a monomial M
$W(M)$	weight of a monomial M
DC	Set of denial constraints
Υ	Set of constraints identifiers
Γ	Set of tuple identifiers
$VC(\mathcal{I}, DC, t)$	Set of constraints violated by a tuple t
$\mathcal{I}^{\Upsilon}$	a Υ-*instance* obtained from $\mathcal{I}$
$\mathcal{I}^{\Upsilon}(R_i)$	A K-*relation* of $\mathcal{I}^{\Upsilon}$
$Q^{k,\alpha}$	top-k query w.r.t. measure α
$ind(R)$	Index associated with R
$label(B)$	Label of a bucket B of an index

Table 2: Summary of the notation used in the paper.

with nondeterministic choices due to the assumption that SBA algorithms do not exploit any specific knowledge on join attributes. Using this framework, we show the optimality of our algorithm w.r.t. the SBA class.

4. PRELIMINARIES

We introduce some basic notions used throughout the paper. Table 2 summarizes the notation. We consider $\mathcal{S} = \{R_1, \dots, R_n\}$ as a database schema with $R_i (i \in [1, n])$ a predicate and with $arity(R_i)$ denoting the arity of R_i. We consider $\mathcal{D}$, $\mathcal{I}$ as the domain and a database instance over $\mathcal{S}$ and $\mathcal{D}$, respectively. Let Γ be an infinite set of identifiers distinct from the domain $\mathcal{D}$. We denote by id a function that associates to each tuple $t \in \mathcal{I}(R_i)$ an unique idenfier $id(t)$ from the set Γ.

Conjunctive Queries (CQ). A CQ is in form $Q(u) \leftarrow R_1(u_1), ..., R_n(u_n)$,where each R_i is a relation symbol in $\mathcal{S}$ and Q is a relation symbol in the output schema, u is a tuple of either distinguished variables or constants and each u_i is a tuple of either variables or constants having the same arity as R_i. The set of variable in a query Q is denoted $Vars(Q)$.

Denial Constraints. A denial constraint is of the form: $\leftarrow R_1(u_1) \wedge ... \wedge R_n(u_n) \wedge \phi$ where the $R_i(u_i)$ are defined as previously and ϕ is a conjunction of comparison atoms of the form x *op* y, where x, y are either constants or variables and *op* $\in \{=, \neq, \leq, <, >, \geq\}$ is a built-in operator. Let C be a denial constraint, C_{id} is an unique identifier of C. We denote Υ the set of the identifiers of the denial constraints.

Monomials and Polynomials. A (non null) 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 $a, m_1, ..., m_n \in \mathbb{N}^*$ (i.e., non zero positive integers) and $x_1, ..., x_n \in \mathcal{X}$. Let $M = a \times x_1^{m_1} \times ... \times x_n^{m_n}$ be a monomial. We denote by $Var(M) = \{x_1, \dots, x_n\}$ 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 $M(P)$ the set of monomials of P.

Provenance Semirings. We recall the provenance semirings framework, introduced in [26] as a unifying framework able to capture a wide range of provenance models at different levels of granularity [26, 25, 20].

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[2] with identities 0 and 1 respectively, $\otimes$ is distributive over $\oplus$ and $0 \otimes a = a \otimes 0 = 0$ holds $\forall a \in K$. An n-ary K-*relation* is a function $R : \mathcal{D}^n \to K$ such that its support, defined by $supp(R) \stackrel{\text{def}}{=} \{t : t \in \mathcal{D}^n, R(t) \neq 0\}$, is finite. 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 [26]. 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 $\mathcal{J}$.

Conjunctive queries on K-*instances*. Let $Q(u) \leftarrow R_1(u_1), ..., R_n(u_n)$, 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: $Q(\mathcal{J}) \stackrel{\text{def}}{=} \{\ (v(u), \Pi_{i=1}^{n} R_i(v(u_i)))\ |\ v \text{ is a valuation over } Vars(Q)\}$.

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: $Q(\mathcal{J})(t) = \sum_{v\ s.t\ v(u)=t} \Pi_{i=1}^{n} R_i(v(u_i)))$.

EXAMPLE 1. *Consider the K-instance hdb^{Υ} in Figure 2 with $K = \mathbb{N}[\Upsilon]$, the set of polynomials with variables from Υ and coefficients in $\mathbb{N}$. The annotation of the tuple $\langle 01, d2, 1\rangle$ of relation S is given by: $hdb^{\Upsilon}(S)(\langle 01, d2, 1\rangle) = C_1$. Evaluation of Q_{ex} over hdb^{Υ} uses $(\mathbb{N}[\Upsilon], +, \times, 0, 1)$ semiring and, for example, leads to an answer $\langle d2, d2\rangle$ annotated as follows: $Q_{ex}(hdb^{\Upsilon})(\langle d2, d2\rangle) = C_1^2 C_2^2 C_3 + C_1^2 C_2^2 C_3^2 + C_1 C_2^2 C_3$.*

5. QUANTIFYING INCONSISTENCY DEGREES OF QUERY ANSWERS

In this section, we focus on the problem of quantifying the inconsistency degree of query answers. Let $\mathcal{I}$ be an instance over a database schema $\mathcal{S}$, let DC be a set of denial constraints over $\mathcal{S}$ and let Υ be the set of identifiers of the constraints in DC. We proceed in three steps:

(i) *Identifying inconsistent base tuples.* We first start by identifying the inconsistent tuples of an instance $\mathcal{I}$ over $\mathcal{S}$ w.r.t a set of denials constraints DC. To achieve this task, we turn each constraint $C \in DC$ into a conjunctive query $Q^{C_{id}}$ and, by exploiting why-provenance, we consider the lineage of $Q^{C_{id}}$ as the set of tuples of $\mathcal{I}$ that violates the constraint C. As a consequence, we are able to define a function $VC(\mathcal{I}, DC, t)$ that associates to each tuple $t \in \mathcal{I}$, the set of constraints violated by t;

(ii) *Annotating the initial database instance.* Using the set

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

$VC(\mathcal{I}, DC, t)$, we convert the instance $\mathcal{I}$ into a Υ-*instance*, denoted $\mathcal{I}^{\Upsilon}$, obtained by annotating each tuple in $\mathcal{I}$ by a monomial with variables from Υ. The variables record the constraints violated by the annotated tuple.

(iii) *Defining inconsistency degrees of query answers.* Given a query Q over the instance $\mathcal{I}$, we use the provenance semirings to annotate the query answers. The latter provenance is the most informative form of provenance annotation [25] and hence is exploited in our setting in order to define two inconsistency degrees for query answers.

We shall detail in the sequel the proposed approach.

5.1 Identifying inconsistency degrees of base tuples

Let $\mathcal{I}$ and DC be respectively an instance and a set of DCs over $\mathcal{S}$. We first convert each constraint C of DC of the form: $\leftarrow R_1(u_1) \wedge ... \wedge R_n(u_n) \wedge \phi$ into a boolean conjunctive query with arithmetic comparisons $Q^{C_{id}}$ defined as follows: $Q^{C_{id}}() \leftarrow R_1(u_1) \wedge ... \wedge R_n(u_n) \wedge \phi$

Example 2. *The set DC of denial constraints depicted in Figure 1 are converted into the following boolean queries:*

$$\begin{array}{lcl} Q^{C_1}() & \leftarrow & D(x,y,z) \wedge S(x,y,u) \wedge z > u \\ Q^{C_2}() & \leftarrow & D(x,y,z) \wedge V(x,y,u) \wedge z > u \\ Q^{C_3}() & \leftarrow & S(x,y,z) \wedge V(x,v,z) \end{array}$$

It is easy to verify that an instance $\mathcal{I}$ violates the set of denial constraints DC iff the boolean UCQ query $Q^{DC} \equiv \bigvee_{C \in DC} Q^{C_{id}}$ evaluates to true over $\mathcal{I}$ (i.e., at least one of the conjunctive queries $Q^{C_{id}}$, for $C \in DC$, returns true when evaluated over $\mathcal{I}$). The base tuples of $\mathcal{I}$ which are inconsistent w.r.t. DC are exactly those tuples of $\mathcal{I}$ that contribute to the computation of the empty answer $<>$ (i.e., true) for query Q^{DC} when evaluated over $\mathcal{I}$. In particular, a base tuple t of $\mathcal{I}$ violates a constraint $C \in DC$ iff t *contributes* to the derivation of the answer true when the query $Q^{C_{id}}$ is evaluated over $\mathcal{I}$.

We shall use below the why-provenance [12] to compute the inconsistent base tuples while keeping track of the constraints violated by each inconsistent tuple. Indeed, the why-provenance of an answer t in a query output is made of the set of all contributing input tuples [12].

We first formulate the why-provenance in the *provenance semirings* framework as proposed in [26]. 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 ($\cup$) 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.

In order to compute the why-provenance, 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 1 (K-instances). *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 $K = \mathcal{P}(\Gamma) \cup \{\bot\}$. The K-instance $\mathcal{I}^{LP}$ 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:*

$$\left\{ \begin{array}{lcll} \mathcal{I}^{LP}(R_i)(t) & = & \{id(t)\} & if\ t \in \mathcal{I}(R_i) \\ \mathcal{I}^{LP}(R_i)(t) & = & \bot & otherwise \end{array} \right.$$

Example 3. *Figure 2 shows the provenance database* hdb^{LP} *obtained from the hospital database* hdb *of our motivating example by annotating each tuple $t \in$* hdb *with a singleton set $\{id(t)\}$ (column lprov) containing the tuple identifier of t.*

Using the provenance semirings $(\mathcal{P}(\Gamma) \cup \{\bot\}, +, ., \bot, \emptyset)$, we define below the function VC that associates with each base tuple t of $\mathcal{I}$ the set of constraints violated by t.

Definition 2 (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: $VC(\mathcal{I}, DC, t) = \{C_{id} \in \Upsilon \mid t \in Q^{C_{id}}(\mathcal{I}^{LP})(<>)\}$*

Example 4. *Consider the boolean conjunctive queries Q^{C_1}, Q^{C_2} and Q^{C_3} of Example 2. Hence, the lineage of the output* true *of each query gives the set of tuples that violate the corresponding constraint:*
$Q^{C_1}(\mathcal{I}^{LP})(\langle\rangle) = \{t_2, t_3, t_4\}$, $\quad Q^{C_2}(\mathcal{I}^{LP})(\langle\rangle) = \{t_2, t_7\}$
$Q^{C_3}(\mathcal{I}^{LP})(\langle\rangle) = \{t_4, t_7\}$

These lineages enable us to compute the set of constraints violated by each tuple t, given by the function $VC(\mathcal{I}, DC, t_1)$. For example, we have $VC(\mathcal{I}, DC, t_1) = \emptyset$, (i.e., t_1 is a consistent tuple) while $VC(\mathcal{I}, DC, t_2) = \{C_1, C_2\}$ and $VC(\mathcal{I}, DC, t_3) = \{C_1\}$.

Note that, even if why-provenance computes richer annotations [12], we choose to not keep the information about combinations of tuples that cause a violation. Hence, our approach is currently agnostic to the *inconsistency reasons.*

5.2 Annotating the database instance

In the next section, we will show how to use the provenance polynomials to define the inconsistency degrees of query answers. In order to obtain that, we first need to convert the instance $\mathcal{I}$ into a $\mathbb{N}[\Upsilon]$-*instance*, denoted $\mathcal{I}^{\Upsilon}$. As shown in the following definition, an instance $\mathcal{I}^{\Upsilon}$ is derived from $\mathcal{I}$ by tagging each tuple $t \in \mathcal{I}$ with a monomial with variables in Υ.

Definition 3 ($\mathcal{I}^{\Upsilon}$ 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 $K = \mathbb{N}[\Upsilon]$. The K-instance $\mathcal{I}^{\Upsilon}$ is constructed as follows: $\forall R_i \in S$ a corresponding K-relation is created in $\mathcal{I}^{\Upsilon}$. A K-relation $\mathcal{I}^{\Upsilon}(R_i) \in \mathcal{I}^{\Upsilon}$ is populated as follows:*

$$\mathcal{I}^{\Upsilon}(R_i)(t) = \left\{ \begin{array}{ll} 0 & if\ t \notin \mathcal{I}^{\Upsilon}(R_i) \\ \prod_{C_{id} \in \Upsilon} C_{id}^{l} & otherwise \end{array} \right.$$

with $l = 1$ if $C_{id} \in VC(\mathcal{I}, DC, t)$ or $l = 0$ otherwise.

Hence, an annotation $\mathcal{I}^{\Upsilon}(R_i)(t)$ 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 $VC(\mathcal{I}, DC, t)$).

PID	RefD	Date	*lprov*	prov
02	$d4$	2	$\{t_1\}$	1
01	$d2$	4	$\{t_2\}$	C_1C_2

Diagnosis (D)

PID	RefD	Date	*lprov*	prov
01	$d2$	1	$\{t_3\}$	C_1
01	$d2$	3	$\{t_4\}$	C_1C_3
02	$d4$	4	$\{t_5\}$	1
01	$d2$	5	$\{t_6\}$	1

Surgery (S)

PID	RefD	Date	*lprov*	prov
01	$d2$	3	$\{t_7\}$	C_2C_3
02	$d4$	3	$\{t_8\}$	1

Vaccination (V)

Figure 2: The K-instances hdb^{LP} (without prov column) and hdb$^{\Upsilon}$ (without lprov column).

Example 5. *Continuing with our example, the hdb$^{\Upsilon}$ instance obtained from the hospital database hdb is depicted in Figure 2. 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: $hdb^{\Upsilon}(R_i)(t_1) = \prod_{C_{id} \in \Upsilon} C_{id}^0 = 1$. For the tuple t_2, we have $VC(\mathcal{I}, DC, t_2) = \{C_1, C_2\}$ and hence: $hdb^{\Upsilon}(R_i)(t_1) = C_1^1 \times C_2^1 \times C_3^0 = C_1C_2$.*

In the sequel, we assume that the relations of an annotated instance $\mathcal{I}^{\Upsilon}$ are augmented with an attribute *prov* that stores the annotations of the base tuples. As an example, Figure 2 shows the annotated relations of the instance *hdb*$^{\Upsilon}$ together with their respective *prov* columns.

5.3 Computing inconsistency degrees of query answers

Given a query Q, we evaluate Q over the $\mathbb{N}[\Upsilon]$-instance $\mathcal{I}^{\Upsilon}$ 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 Υ of constraint, are fairly informative as they allow to fully record how the constraints are violated by base tuples that contribute to each answer. Such annotations are hence exploited to compute the various inconsistency measures needed for query answers.

Example 6. *Continuing with the example, evaluating the query Q_{ex} over hdb$^{\Upsilon}$ and computing its polynomial provenance leads to the following annotated answers:*
$Q_{ex}(hdb^{\Upsilon})(\langle d2, d2\rangle) = C_1^2C_2^2C_3 + C_1^2C_2^2C_3^2 + C_1C_2^2C_3$
$Q_{ex}(hdb^{\Upsilon})(\langle d4, d4\rangle) = 1$.
The monomial $C_1^2C_2^2C_3$ 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 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 $Q(\mathcal{I}^{\Upsilon})(t)$ fully records the inconsistency of an output t in terms of violations of constraints and therefore can be used to quantify the inconsistency degrees of a query outputs. Consider a polynomial $P = Q(\mathcal{I}^{\Upsilon})(t)$ of an output t of a given query Q. Each monomial M from P gives an alternative way to derive the output t. In the sequel, we consider bag semantics of query answers. Although our approach is extensible to set semantics of query answers, we do not discuss this further in the paper.

Under bag semantics, each query answer will be annotated with a monomial corresponding to the unique derivation of the considered answer.

Two different measures can be defined in order to quantify the inconsistency degree of an answer t depending on how one deals with multiple occurrences of the same variable in monomials. This situation occurs when a constraint is violated by more than one base tuple that contribute to an answer t. We define two possible quantifications to deal with this issue: *single occurrence quantification*, in which a variable that appears in a monomial is counted exactly once, and *multi-occurrence quantification*, where the exact number of occurrences of a variable in a monomial is taken into account when quantifying the inconsistency degree of a given answer. As a consequence, we obtain two different inconsistency measures: the CBM (Constraint-based, Bag semantics, Multiple occurrence) measure and the CBS (Constraint-based, Bag semantics, Single occurrence) measure.

Definition 4 (Inconsistency measures). *Let $\mathcal{I}$, Q and DC be defined as previously. Let $M = Q(\mathcal{I}^{\Upsilon})(t)$ be the monomial annotating an output t of Q. We define inconsistency measures of t as: $CBM(t, Q, \mathcal{I}, DC) \overset{def}{=} W(M)$ and $CBS(t, Q, \mathcal{I}, DC) \overset{def}{=} |Var(M)|$*

It follows that a single occurrence quantification of a monomial M amounts to counting the number of distinct variables that occur in M while a multi occurrence semantics sums the total number of occurrence of each variable in M (given by the weight $W(M)$ of M).

Example 7. *Consider the annotation*
$Q_{ex}(hdb^{\Upsilon})(\langle d2, d2\rangle) = \underbrace{C_1^2C_2^2C_3}_{M_1} + \underbrace{C_1^2C_2^2C_3^2}_{M_2} + \underbrace{C_1C_2^2C_3}_{M_3}$.
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}. 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 output three times leading to three answers, $a_1 = a_2 = a_3 = \langle d2, d2\rangle$, each of which annotated, respectively, whith one of the monomials M_1, M_2 and M_3. 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 $CBM(a_3, Q, \mathcal{I}, DC) = 4$.

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 1. *Let $\mathcal{I}, Q, DC$ be defined as previously. $\forall t \in Q(\mathcal{I})$ we have:*
$CBM(t, Q, \mathcal{I}, DC) = CBS(t, Q, \mathcal{I}, DC) = 0 \Rightarrow t$ is a CQA.

Lemma 1 is trivial 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. More differences between CQA and our method are addressed in Section 7.

6. INCONSISTENCY-AWARE RANKING

In this section, we study top-k ranking of inconsistency-awre tuples as defined above. To this end, we leverage the introduced inconsistency measures. We consider as input an annotated instance $\mathcal{I}^{\Upsilon}$, where each base tuple t of a relation R is annotated with the monomial $\mathcal{I}^{\Upsilon}(R)(t)$ (c.f. Definition 3). Let α be either CBM or CBS, 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 $CBM(t, Q, \mathcal{I}, DC) = W(M)$ (respectively, $CBS(t, Q, \mathcal{I}, DC) = |Var(M)|$), with $M = Q(\mathcal{I}^{\Upsilon})(t)$. 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.

Definition 5 (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 instance. Let k be an integer, let $\alpha \in \{CBM, CBS\}$. The top-k query answers of a query Q using the inconsistency measure α, denoted by $Q^{k,\alpha}(\mathcal{I})$, is defined as follows:*
***(i)** $Q^{k,\alpha}(\mathcal{I}) \subseteq Q(\mathcal{I})$,*
***(ii)** $|Q^{k,\alpha}(\mathcal{I})| = Min(k, |Q(\mathcal{I})|)$, and*
***(iii)** $\forall(t_1, t_2) \in Q^{k,\alpha}(\mathcal{I}) \times (Q(\mathcal{I}) \setminus Q^{k,,\alpha}(\mathcal{I}))$, we have: $\alpha(t_1, Q, \mathcal{I}, DC) \leq \alpha(t_2, Q, \mathcal{I}, DC)$*

Condition *(i)* and *(ii)* ensure that a top-k query $Q^{k,\alpha}(\mathcal{I})$ 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 following example illustrates a top-k query over our running example.

Example 8. *Assume $k = 1$ and $\alpha = CBM$. Continuing with the database **hdb** and the query Q_{ex} as in Figure 1, we have: $Q^{k,\alpha}_{ex}(\textbf{hdb}) = \{\langle d4, d4\rangle\}$.*

A naive approach to solve a top-k query $Q^{k,\alpha}(\mathcal{I})$, with α being one of our inconsistency measures, would be to entirely compute and sort, w.r.t. α, the set $Q(\mathcal{I})$ and then filter out the k best answers. Such a naive approach is clearly suboptimal in particular when $k \ll |Q(\mathcal{I})|$.

The problem of computing $Q^{k,\alpha}(\mathcal{I})$ answers falls in the general setting of rank join problems [28]. Performance of rank join algorithms have been deeply studied in the case of monotonic scoring functions [28, 39] while very few works deal with nonmonotoninc functions.

Let us first recall the monotonicity property. Let f be a scoring function. The function f is monotonic w.r.t. an operator $\circ$ iff: $\mathsf{f}(x) \leq \mathsf{f}(y) \wedge \mathsf{f}(z) \leq \mathsf{f}(v) \Rightarrow \mathsf{f}(x \circ z) \leq \mathsf{f}(y \circ v)$, $\forall x, y, z, v$. Otherwise, f is said to be non-monotonic w.r.t. $\circ$.

The following lemma says that inconsistency degrees based on *multi occurrence* quantification are monotonic w.r.t. the join operator ($\bowtie$) while the inconsistency measures based on the *single occurrence* quantification are non-monotonic w.r.t. $\bowtie$.

Lemma 2. *A scoring function that associates to each tuple t a score computed using CBM (respectively, using CBS) is monotonic (respectively, non-monotonic) w.r.t. $\bowtie$.*

We present in the next section an integrated algorithm to handle both monotonic and non-monotonic inconsistency measures in the case of bag semantics and we prove the optimality of the algorithm.

6.1 The 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 rank jon algorithms [28] 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 9. *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 $|label(B)| \leq |label(B')|$.

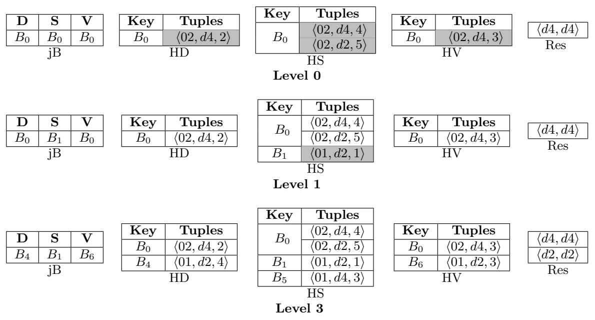
Example 10. *The following indexes are associated with the relations D, S and V of our example database: $Ind(D) = [B_0, B_4]$, $Ind(V) = [B_0, B_6]$ and $Ind(S) = [B_0, B_1, B_5]$.*

We now describe the pseudocode of 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_n (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
             $Q^{k,\alpha}$ : a top-k query over $R_1, \ldots, R_m$ with
    $\alpha \in \{CBM, CBS\}$
             $ind(R_1), \ldots, ind(R_m)$ : indexes of the input
             relations
    Output: Res: k best answers w.r.t. $\alpha$
    Data structures: $HR_1, \ldots, HR_m$: 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 $l \leq |ViolC| \wedge Res.size < k$ do
6           curVSet := $\emptyset$ ;
7           IterateJoin(1, level, curVSet) ;
8           level:= level + 1 ;
9       return $Res$
```

EXAMPLE 11. *We assume that the input relations are processed in this order: D, S and V. We illustrate the processing of query $Q^{2,CBS}_{ex}(\mathcal{I})$ by TopINC. Figure 3 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 3. 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 3. 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) 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*



Level 0
jB: D S V / B_0 B_0 B_0
HD: Key Tuples / B_0 $\langle 02, d4, 2\rangle$
HS: Key Tuples / B_0 $\langle 02, d4, 4\rangle$ $\langle 02, d2, 5\rangle$
HV: Key Tuples / B_0 $\langle 02, d4, 3\rangle$
Res: $\langle d4, d4\rangle$
Level 1
jB: D S V / B_0 B_1 B_0
HD: Key Tuples / B_0 $\langle 02, d4, 2\rangle$
HS: Key Tuples / B_0 $\langle 02, d4, 4\rangle$ $\langle 02, d2, 5\rangle$ / B_1 $\langle 01, d2, 1\rangle$
HV: Key Tuples / B_0 $\langle 02, d4, 3\rangle$
Res: $\langle d4, d4\rangle$
Level 3
jB: D S V / B_4 B_1 B_6
HD: Key Tuples / B_0 $\langle 02, d4, 2\rangle$ / B_4 $\langle 01, d2, 4\rangle$
HS: Key Tuples / B_0 $\langle 02, d4, 4\rangle$ $\langle 02, d2, 5\rangle$ / B_1 $\langle 01, d2, 1\rangle$ / B_5 $\langle 01, d4, 3\rangle$
HV: Key Tuples / B_0 $\langle 02, d4, 3\rangle$ / B_6 $\langle 01, d2, 3\rangle$
Res: $\langle d4, d4\rangle$ $\langle d2, d2\rangle$

Figure 3: Illustraive example of TopINC

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 $i \leq idx.size \wedge |label(idx[i])| \leq level$ do
6           if $\alpha = CBS$ then
7               $curVSet := curVSet \cup label(idx[i])$
8           else
9               /* $\uplus$ stands for bag union                   */
10              $curVSet := curVSet \uplus label(idx[i])$
11          /* Case input $R_p$ is not the last                  */
12          if $p < m \wedge |curVSet| <= level$ then
13              jB[p]:=idx[i];
14              $IterateJoin(p + 1, level, curVSet)$; ;
15          if $p = m \wedge |curVSet| = level$ then
16              jB[p]:=idx[i];
17              /* Compute the join from jB                      */
18              $ans := HR_1(jB[1]) \bowtie \ldots \bowtie HR_m(jB[m])$ ;
19              /* Add the results to Res up to k                */
20              $Res.add(ans)$ ;
21              if $Res.size = k$ then
22                  EXIT;
23          $i := i + 1$;
```

$jB = [B_0, B_1, B0]$, 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 3 and halts when it reaches $|res| = 2$.

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

LEMMA 3. *Let $\alpha \in \{CBM, CBS\}$. The algorithm TopINC computes correctly $Q^{k,\alpha}(\mathcal{I})$.*

LEMMA 4. *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 $O(|\mathcal{I}| + k \times s)$.*

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 [28, 39]), 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 missing proofs due to space limitations can be found in [30].

Optimality of TopINC. As discussed in Section 3, the presence of the non-monotonic function CBS prevents us from exploiting the existing frameworks [31, 39] to analyze the performance of TopINC. We define hereafter a new class of algorithms, called Semi-Blind Algorithms (SBA), which includes the algorithms that exploit only the *prov* column (i.e., the scoring function) without imposing any predefined data access strategy[3]. We show that in this general class, and modulo non deterministic choices, TopINC is optimal w.r.t. the number of tuples read from the inputs.

We start by defining the family of SBA algorithms as follows. 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 $label(J_i) \overset{\text{def}}{=} \left(\pi_{prov}(t_1^i), \ldots, \pi_{prov}(t_m^i)\right)$. 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 6 (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. 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 $cost(Al, Q, \mathcal{I})$.

[3]Note that the popular family of top-k algorithms using sorted data access [22, 28, 31] is a strict subset of SBA.

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: $cost(Al, Q, \mathcal{I}) = |\bigcup_{i=1}^{n} tuple(J_i)|$. The following lemma states that there exists no algorithm in the class SBA that is optimal w.r.t. the *cost* metric defined above.

Lemma 5. *Let $Q^{k,\alpha}$ be a top-k query. Then, we have: $\forall Al_1 \in SBA$, $\exists Al_2 \in SBA$ and an instance $\mathcal{I}$ such that, $cost(Al_1, \mathcal{I}, Q) > cost(Al_2, \mathcal{I}, Q)$.*

The intuition behind the above lemma 5 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 subsequences 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 $[J_1, ..., J_n]$, 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 $\mathcal{I}$: $cost^{\nabla}(Al, Q, \mathcal{I}) = |Regs(Al, Q, \mathcal{I})|$.

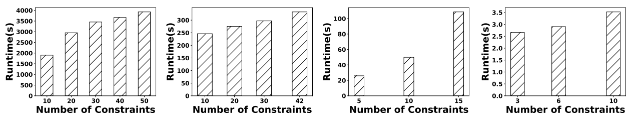
The introduced cost model $cost^{\nabla}(Al, Q, \mathcal{I})$ conveniently prevents an algorithm to compute an answer that can be dropped after to the top-k answers, thus avoiding more useless I/O operations, as proved in the following theorem.

Theorem 1. *For any instance $\mathcal{I}$ and any top-k conjunctive query $Q^{k,\alpha}$, we have: $cost^{\nabla}(\textsf{TopINC}, Q, \mathcal{I}) \leq cost^{\nabla}(AL, Q, \mathcal{I})$, $\forall AL \in SBA$.*

Notice that the TopINC algorithm is only sensitive to the class of queries and unsensitive to the class of constraints, the latter being hardcoded in the inconsistency an-

Dataset	Syn	#Rel	#Tup	#Cons		#atom	Inc(%)
				#Equal	#(In-)equal		
Hospital	✗	1	114919	3	39	2	100
Tax	✗	1	99999	1	49	2	100
Synthetic	✓	5	1012524	6	9	[1, 3]	89.34
Pstock	✗	1	244992	1	9	[1, 2]	18.62

Table 3: Datasets used in our empirical evaluation.



(a) Tax (b) Hospital (c) Synthetic (d) Pstock

Figure 4: Transformation of an instance into a $\mathbb{N}[\Upsilon]$-instance

notations. TopINC and Theorem 1 are established for conjunctive queries, which are by definition monotonic queries (i.e. for which a partial output can be computed starting from a partial input). Non-monotonic queries (such as aggregate queries and queries with negation) are not covered by our algorithm and are far from being trivial due to the non-monotonic nature of the *CBS* scoring function.

7. EMPIRICAL EVALUATION

Our experimental assessment has been carried out with two main objectives in mind. In the first part, we aimed at measuring the time required to transform a database instance into a $\mathbb{N}[\Upsilon]$-*instance*. The latter transformation is considered as the pre-processing time for our approach. In second part, we empirically evaluate the TopINC algorithm to perform top-k query answering. 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. The datasets used in our study are summarized in Table 3. As real-life datasets, we employed the Hospital, Tax and Pstock datasets from [17] as they are the only datasets in the literature that are equipped with denial constraints. In addition, we generated our own synthetic datasets and companion denial constraints.

In Table 3, 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.

Table 3 shows the constraints in each dataset. Columns #Equal and #(In-)equal provide the number of constraints that use exclusively equality predicates and those using a mixture of predicates from $\{\leq, \geq, \neq, <, >, =\}$, respectively. As one can notice, the latter are in greater number.

Runtimes of the instance transformation. In this Section, we measure the runtimes of the transformation of a database instance into a $\mathbb{N}[\Upsilon]$-*instance*. Figure 4 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 4.a) and Hospital (Figure 4.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).

	Query	#Answers	CBM	CBS
Q_1	$1\ s$	2891897	$119\ s$	$59\ s$
Q_2	$214ms$	560 161	$22\ s$	$11\ s$
Q_3	$41\ ms$	114 919	$2\ s$	$2\ s$
Q_4	$31ms$	50	$3\ ms$	$2\ ms$
Q_5	$57\ ms$	625	$31ms$	$19ms$
Q_6	$23\ ms$	99 999	$546ms$	$546ms$
Q_7	$97\ ms$	100 777	$903s$	$558ms$
Q_8	$518ms$	488 505	$6\ s$	$4\ s$
Q_9	$61\ ms$	303	$83ms$	$80ms$
Q_{10}	$231ms$	582	$12ms$	$12ms$
Q_{11}	$472ms$	505 046	$7\ s$	$4\ s$
Q_{12}	$3\ s$	14831052	$5\ mn$	$2\ mn$
Q_{13}	$1\ s$	7384207	$2\ mn$	$1\ mn$
Q_{14}	$938ms$	287 242	$9\ s$	$3\ s$

Across all tuples in the query output
(a)

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

One tuple at a time
(b)

Figure 5: CBS and CBM computation overhead.

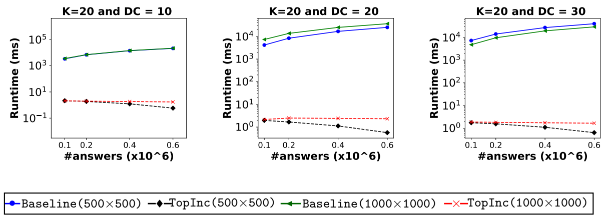


Figure 6: TopINC vs. Baseline ($\alpha = CBS$)

Overhead of inconsistency measures on query execution. 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 5.a) as well as the total overhead for the complete output of the query (Table 5.b). The obtained results are reported in Table 5. 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 5.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 $2ms$ and $6m$, respectively for queries Q_4 and Q_{12}. (respectively, yellow and red cells in Table 5.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 returns the top-k results tuple by tuple. Thus, one should look at the overhead per tuple in Table 5.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 5.b).

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 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.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. In Figure 7, from 7.f to 7.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 6. 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 6.a, 6.b and 6.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.

Qualitative evaluation. 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 in [1, 37] and containing census data, along with and Food Inspection [2] including information about inspections of food establishments in Chicago. The features of the two datasets are shown in Table 4.a. Table 4.b reports the the constraints of Adult, namely A_1 and A_2, that have been derived using Holoclean [38]. While A_1 indicates that men who have 'married' as marital status are husbands, A_2 expresses the dual constraint for women.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 4.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."*

Dataset	Table	#tuples	Attributes
Adult	Adult	48842	11
Food Inspection	Inspection	204896	17

(a) Characteristics of Adult and Food Inspection.

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

(b) Constraints on Adult.

F_1:←	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:←	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:←	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 ≠ '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 4: Real-world datasets with their denial constraints.

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 5: Results of the qualitative study.

We report in Table 4.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

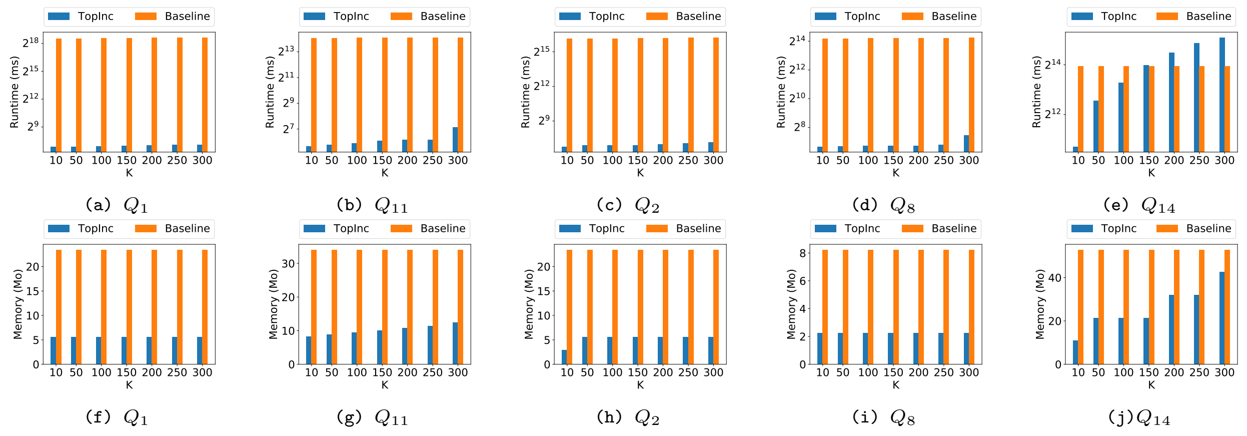


Figure 7: TopINC performance vs baseline (α = CBS). (a)-(e) Runtime, (f)-(j) Memory footprint

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 5.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 5.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 5.b). These tuples are captured when running TopINC for top-100 tuples starting from the most inconsistent ones as illustrated in Table 5.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 5.c), the CBS measure provides a different ranking for the 23 answers of the 4th row of this Table.

We show in Table 5.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. Furthermore, by looking at the tuples that contribute to the last row of Table 5.d ($CBM = 9$) violating the three consraints $F_1, F2$ and F_3 in tandem, we made an interesting discovery. These violations (and also those corresponding to $CBM = 8$) are all related to the same erroneous license numbers (all set to 0 in the corresponding tuples).

Finally, we show by means of examples the remarkable difference between our approach and CQA[4]. Note that as we already pinpointed in Section 3, these are complementary approaches. Table 5.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 5.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 5.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.

8. CONCLUSION AND OUTLOOK

We have presented a novel framework for inconsistency-aware query answering leveraging two different measures of inconsistency. We have grounded the computation of inconsistency degrees in why-provenance annotations and we have designed a novel top-k rank algorithm suitable for the above measures while proving its optimality.

As future work, we plan to investigate the extension of our approach to other classes of constraints and queries, such as universal constraints and aggregate queries. Techniques developed in the literature e.g. [18] for tracing lineage of first order queries could be employed to extend our approach to the upper class of constraints beyond DCs (e.g., universal constraints). However, in the presence of negation, the connection between provenance and inconsistency degrees of base tuples remains unclear and raises intriguing research questions. On the other hand, while recent literature, e.g., [4], provides foundations to study the computation of inconsistency measures for aggregate queries using polynomial provenance, the problem of efficiently computing top-k aggregate queries in presence of non-monotonic scoring functions such as CBS remains open and unsolved up to date.

Handling updates of tuples or denial constraints in our framework is also in our research agenda. One has to resort to classical incremental view maintenance mechanisms. However, handling constraint modifications without fully recomputing both the constraint-based annotations and the TopINC's index, remains an open challenge.

9. ACKNOWLEDGEMENTS

Our research is supported by the French National Agency (ANR) through the grant nr. 18-CE23-0002 QualiHealth.

[4]We consider repair by deletion and symmetric set difference as measure of minimality [5].

10. REFERENCES

[1] Adult dataset. `https://github.com/HoloClean/holoclean/blob/master/testdata/AdultFull.csv`.

[2] Food inspection dataset. `https://data.cityofchicago.org/Health-Human-Services/Food-Inspections/4ijn-s7e5`.

[3] S. Amer-Yahia, S. Elbassuoni, A. Ghizzawi, R. M. Borromeo, E. Hoareau, and P. Mulhem. Fairness in online jobs: A case study on taskrabbit and google. In *EDBT 2020*, pages 510–521, 2020.

[4] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *ACM PODS 2011*, page 153–164, 2011.

[5] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *ACM PODS 1999*, pages 68–79, 1999.

[6] A. Arioua and A. Bonifati. User-guided repairing of inconsistent knowledge bases. In *EDBT 2018*, pages 133–144, 2018.

[7] L. Bertossi. *Database Repairing and Consistent Query Answering*. Morgan & Claypool, 2011.

[8] L. Bertossi. Database repairs and consistent query answering: Origins and further developments. In *ACM PODS*, page 48–58, 2019.

[9] L. Bertossi, A. Hunter, and T. Schaub. Introduction to inconsistency tolerance. In *Inconsistency Tolerance*, volume LNCS 3300, pages 1–14, 2005.

[10] L. E. Bertossi. Repair-based degrees of database inconsistency: Computation and complexity. *CoRR*, abs/1809.10286, 2018.

[11] L. E. Bertossi and J. Chomicki. Query answering in inconsistent databases. In J. Chomicki, R. van der Meyden, and G. Saake, editors, *In Logics for Emerging Applications of Databases*, 2013.

[12] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT 2001*, 2001.

[13] M. Calautti, M. Console, and A. Pieris. Counting database repairs under primary keys revisited. In *ACM PODS*, page 104–118, 2019.

[14] A. Calì, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *ACM PODS 2003*, pages 260–271, 2003.

[15] L. Chengkai, C.-C. C. Kevin, and I. Ihab F. Supporting ad-hoc ranking aggregates. In *ACM SIGMOD 2006*, 2006.

[16] J. Chomicki, J. Marcinkowski, and S. Staworko. Computing consistent query answers using conflict hypergraphs. In *CIKM 2004*, pages 417–426, 2004.

[17] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, Aug. 2013.

[18] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM TODS*, 25(2):179–227, 2000.

[19] H. Decker and D. Martinenghi. Modeling, measuring and monitoring the quality of information. In *ER 2009 Workshops*. ACM, 2009.

[20] D. Deutch, T. Milo, S. Roy, and V. Tannen. Circuits for datalog provenance. In *ICDT 2014*, 03 2014.

[21] D. Didier, L. Jerôme, and P. Henri. Possibilistic logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 439–513. Oxford University Press.

[22] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *ACM PODS*, page 102–113, 2001.

[23] F. Geerts, F. Pijcke, and J. Wijsen. First-order under-approximations of consistent query answers. *Int. J. Approx. Reasoning*, 83(C):337–355, Apr. 2017.

[24] J. Grant and A. Hunter. Measuring inconsistency in knowledgebases. *Journal of Intelligent Information Systems*, 2005.

[25] T. J. Green. Containment of conjunctive queries on annotated relations. In *ICDT*, pages 296–309. ACM, 2009.

[26] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *ACM PODS 2007*, pages 31–40. ACM, 2007.

[27] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB Journal*, 13(3):207–221, Sept. 2004.

[28] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 2008.

[29] I. F. Ilyas and X. Chu. *Data Cleaning*. ACM, 2019.

[30] O. Issa, A. Bonifati, and F. Toumani. Evaluating Top-k Queries with Inconsistency Degrees. `https://hal.archives-ouvertes.fr/hal-02898931`. 2020.

[31] N. P. Karl Schnaitter. Evaluating rank joins with optimal cost. In *ACM PODS 2008*, pages 43–52, 2008.

[32] S. Kolahi and L. V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT 2009*, pages 53–62, 2009.

[33] J. Lang and P. Marquis. Reasoning under inconsistency: A forgetting-based approach. *Artif. Intell.*, 174(12-13):799–823, 2010.

[34] E. Livshits and B. Kimelfeld. Counting and enumerating (preferred) database repairs. In *ACM PODS*, pages 281–301, 2017.

[35] E. L. Lozinskii. Information and evidence in logic systems. *Journal of Experimental and Theoretical Artificial Intelligence*, pages 163–193, 1994.

[36] D. Maslowski and J. Wijsen. A dichotomy in the complexity of counting database repairs. *Journal of Computer and System Sciences*, 79(6):958 – 983, 2013.

[37] J. Rammelaere and F. Geerts. Explaining repaired data with CFDs. *PVLDB*, 11(11):1387–1399, 2018.

[38] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[39] K. Schnaitter and N. Polyzotis. Optimal algorithms for evaluating rank joins in database systems. *ACM TODS*, 35(1):6:1–6:47, 2010.

[40] J. Wijsen. Database repairing using updates. *ACM TODS*, 30(3):722–768, 2005.

[41] D. Xin, J. Han, and K. C.-C. Chang. Progressive and selective merge: computing top-k with ad-hoc ranking functions. In *ACM SIGMOD*, pages 103–114, 2007.