IJESRT
INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Consistent Query Answering in Inconsistent Databases

Zanamwe Ngonidzashe
Department of Computer Science, University of Zimbabwe, Zimbabwe
nbzanamwe@gmail.com

Abstract
These days it is more common to build integrated or federated databases that draw data from multiple, autonomous and distributed sources. The problem of data integration is nevertheless very complex. This paper concentrates on consistent query answering, which is a specific issue arising in data integration. Consistent query answering is an approach to retrieving consistent answers over databases that might be inconsistent with respect to some given integrity constraints. The approach is based on concepts of repair and query rewriting. This paper surveys several recent researches on obtaining consistent information from inconsistent databases, such as the underlying query rewriting model and a number of approaches to computing consistent query answers.

Introduction
This paper focuses on query answering in inconsistent databases. Arenas, Bertossi, and Chomicki (1999) define an inconsistent database as one that violates a given set of integrity constraints. So the major thrust of this paper is to explore how *consistent* information can be obtained from *inconsistent* databases in response to user queries. Arenas, Bertossi, and Chomicki (1999) establish that the traditional practice with regards to inconsistent databases is to back out transactions that violate integrity constraints. However this paper does not go by traditional practice. Instead, the paper presents ways of querying inconsistent databases such as repairs, query rewriting, consistent answers, ConQuer, logic programs and so on.

Bertossi and Chomicki (2003) indicate that a *consistent query answer* is, intuitively, true regardless of the way the database is fixed to remove integrity constraint violations. Thus answer consistency serves as an indication of its reliability. On the other hand, Arenas et al (1999) note that an answer to a query is consistent if it is obtained as an answer every time the query is posed to a minimally repaired version of the original database. Intuitively, an answer to a query posed to a database that violates the integrity constraints will be consistent in a precise sense: It should be the same as the answer obtained from any minimally repaired version of the original database. Bertossi, and Chomicki (2003) further assert that a database repair is another database that is *consistent* and *minimally* differs from the original database. In this paper, only consistent query answers for first-order and scalar aggregation queries will be considered.

Rahm and Do (2000) aver that, although there exist a wide variety of tools for automatic elimination of duplicates, extraction and standardization of information, there are practically no tools that automatically resolve integrity constraint violations. Further, they affirm that, the user is responsible for providing a procedure that decides how the conflicts should be resolved. Vassiliadis, Vagena, Skiadopoulos, and Karayannidis (2000) establish that the standard repertoire of actions that can be performed on a conflicting tuple is: removing the tuple, leaving the tuple, or reporting the tuple to an auxiliary table.

1.1 Inconsistent data scenarios
This section covers practical scenarios where data is inconsistent with integrity constraints. Bertossi, and Chomicki (2003) present the following scenarios:

1.1.1 Integration of autonomous data sources.
Integration of independent data sources is one of the causes of inconsistency in databases. It is possible and common that independent data sources may separately satisfy given integrity constraints, but when they are integrated together the constraints may be violated. For instance, consider different, conflicting salary for the same employee in an employee database and a taxpayer database. Each of those databases separately satisfies the functional dependency that associates a single salary with each employee, and yet together they violate this dependency. Additionally, since the sources are autonomous, the violations cannot be simply fixed by removing one of the conflicting tuples.

1.1.2 Unenforced integrity constraints.

Even though integrity constraints capture an important part of the semantics of a given application, they may still fail to be enforced for a variety of reasons. A data source may be a legacy system that does not support the notion of integrity checking altogether. Or, integrity checking may be too costly (this is often the reason for dropping some integrity constraints from a database schema). Finally, the database management system (DBMS) itself may support only a limited class of constraints.

1.1.3 Temporary inconsistencies.

It may often be the case that the consistency of a database is only temporarily violated, and further updates or transactions are expected to restore it. This phenomenon is becoming more and more common, as the databases are increasingly involved in a variety of long-running activities or workflows.

1.1.4 Conflict resolution.

Removing tuples from a database to restore consistency leads to information loss, which may be undesirable. For example, one may want to keep multiple phone numbers for an employee if it is not clear which is the correct one. In general, the process of conflict resolution may be complex, costly, and non-deterministic. In real-time decision-making applications, there may not be enough time to resolve all conflicts relevant to a query.

In line with the above, Arenas, Bertossi, and Chomicki (1999) assert that databases may become inconsistent with respect to a given set of integrity constraints because of the following factors:

1. Certain integrity constraints cannot be expressed or maintained by existing database management systems.
2. Transient inconsistencies caused by the inherent non-atomicity of database transactions.
3. Delayed updates of data warehouses.
4. Integration of heterogeneous databases particularly with duplicated information.
5. Inconsistency with respect to soft integrity constraints where transactions in violation of their conditions are not prevented from executing.
6. Legacy data on which one wants to impose constraints. The consistency of the database will be restored by executing further transactions.
7. User constraints that cannot be checked or maintained

1.2 Applications of consistent query answering techniques

This section looks at types of databases where data inconsistencies are common and also where solutions to querying inconsistent databases can be applied.

1.2.1 Data warehousing.

Data contained in a data warehouse comes from various sources and as such, some of it typically may not satisfy the given integrity constraints. The normal approach is to *clean* the data by removing inconsistencies before the data is stored in the warehouse. However, a different scenario becomes possible wherein the inconsistencies are not removed but rather query answers are marked as "consistent" or "inconsistent". In this way, information loss due to data cleaning may be prevented.

1.2.2 Database integration.

Often, many different databases are integrated together to provide a single unified view for the users. Database integration is difficult since it requires the resolution of many different kinds of discrepancies of the integrated databases. One possible discrepancy is due to different sets of integrity constraints. Moreover, even if every integrated database *locally* satisfies the same integrity constraint, the constraint may be *globally* violated. For example, different databases may assign different addresses to the same student. Such conflicts may fail to be resolved at all and inconsistent data cannot be "cleaned" because of the autonomy of different databases. Therefore, it is important to be able to find out, given a set of local integrity constraints, which query answers returned from the integrated database are consistent with the constraints and which are not.

1.1.3 Active and reactive databases.

A violation of integrity constraints may be acceptable under the provision that it will be repaired in the near future. For example, the stock level in a warehouse may be allowed to fall below the required minimum if the necessary replenishments have been ordered. During this temporary inconsistency, however, query answers should give an indication whether they are consistent with the constraints or not. This problem is particularly acute in active databases that allow such consistency lapses. The result of evaluating a trigger condition that is consistent with the integrity constraints should be treated differently from the one that isn't.

1.3 Classes of Integrity constraints

Integrity constraints are defined as typed, closed first-order language formulas. Xie and Yang (2007) note that, integrity constraints (ICs) maintain the consistency and validity of data and enable them to conform to the rules of entities in the real world effectively. Chomicki, Marcinkowski and Staworko

(2004), note that the main role of integrity constraints in databases is to enforce consistency. The occurrence of integrity violations is prevented by DBMS software. The following are the classes of integrity constraints:

1. Universal integrity constraints: are constraints that apply to an entire database schema
2. Denial constraints: They are a special case of universal constraints. Jan Chomicki and Jerzy Marcinkowski (2000) establish that denial constraints allow an arbitrary number of literals per constraint and arbitrary built-in predicates. Further, they comment that denial constraints also relax the typedness restriction of functional dependencies and are particularly useful for databases with interpreted data, e.g., numbers.
3. Binary constraints: universal constraints with at most two occurrences of database relations.
4. Functional dependencies (FDs): They are a special case of binary denial constraints. A more familiar formulation of a FD is X → Y where X is the set of attributes of P corresponding to x1 and Y the set of attributes of P corresponding to x2.
5. Referential integrity constraints, also known as inclusion dependencies: these are constraints that stipulate that values of all foreign keys must be consistent with values of referencing relations.

Solutions for Inconsistency Handling

2.1 Paraconsistent logics (Hunter, 1998 and Grant and Subrahmanian, 2000)

2.2 Non-repairing and merging-oriented techniques:

a. Pre-orders on information sets (Cantwell, 1998 or Marquis and Porquet, 2003 in the paraconsistent framework).
b. Argumentative hierarchy (Elvang-Goransson and Hunter, 1995), argumentative frameworks (Dung, 1995) and databases (Pradhan, 2003).
c. Fusion rules (Appriou et al., 2001).
d. Merging databases (Cholvy and Moral, 2001).
e. Contextualizing ontologies (Bouquet, Giunchiglia, van Harmelen, Serafini and Stuckenschmidt, 2003) and data (MacGregor and Ko, 2003).

2.3 Measuring the anomalies:

a. Evaluating by means of paraconsistent logic (Hunter, 2003).
b. Measuring inconsistent information (Knight, 2003).
c. Consistent interpretation of Skolem noise (Alonso et al., 2003).

2.4 Repairing techniques:

a. To apply knowledge reductions in inconsistent systems (Kryszkiewiccz, 2001).
b. Fellegi-Holt method (Boskovitz, Goré and Hegland, 2003).
c. Database repairs by tableaux method (Bertossi and Schwind, 2004).
d. Consistent querying to repair databases (Greco and Zumpano, 2000).
e. Consistent enforcement of the database by means of greatest consistent specializations (Link, 2003).

2.5 Consistent answering techniques without reparation:

a. Transformation of the query to obtain consistent answers (Celle and Bertossi, 1994).
b. Consistent query answer in the presence of inconsistent databases (Greco and Zumpano, 2000)
c. To use bounded paraconsistent inference (see e.g. Marquis and Porquet, 2003).
d. Detecting the cause of the inconsistency and retrieving a subset of the original knowledge database (Arieli and Avron, 1999).

2.6 Consistency preserving methods:

a. Consistency preserving updates in deductive databases (Mayol and Teniente, 2003).

The above six methods and their respective categories can be applied when querying inconsistent databases. Due to time and other constraints it is impractical to explore all the methods in this paper hence attention will be focused on method 4 and 5. The reason is that, they are much more found in practice than other methods. The next sections are devoted to detailed discussions of the two approaches namely repairing databases and consistent answering techniques without reparation.

Database Repairs

Arenas, Bertossi and Chomicki (1999) define a database repair (hereinafter referred to as a repair) as an instance of the same database schema that does satisfy the integrity constraints and differs from the original instance by a minimal set of changes. Depending on what is meant by minimal set of changes, different repair semantics can be obtained. In line with the above, Caniup´an and Bertossi (2007) indicate that a repair is a database instance obtained from database D by deleting or inserting whole tuples and that repair satisfies the ICs and minimally differs from D (under set inclusion). Further, Bertossi (2006) establishes that a tuple *t* is a consistent answer to query *Q* in a database instance r with respect to integrity constraints *IC* whenever *t* is an answer to *Q* in every repair of *r*. Also, Bertossi (2006) asserts that a repair must meet the following conditions, a repair of a database instance r is a database instance r_0 over the same schema and domain, satisfies *IC* and differs from *r* by a minimal set of changes (insertions/deletions of whole tuples). For example, consider the following relational database instance Student:

STUDENT

Name	Mark
Zanamwe	50
Zanamwe	75
Madzima	76
Musara	67

Here the query: *Select * from Student* has two consistent answers: Madzima, 76 and Masara, 67

The instance Student violates the functional dependency F1: Name → Mark through the first two tuples. This is an inconsistent database. Nevertheless, there is still some consistent information in it. For example, only the first two tuples participate in the integrity violation. In order to characterize the consistent information, we notice that there are two possible ways to repair the database in a minimal way if only deletions and insertions of whole tuples are allowed. They give rise to two different repairs:

STUDENT1

Name	Mark
Zanamwe	50
Madzima	76
Musara	67

STUDENT2

Name	Mark
Zanamwe	75
Madzima	76
Musara	67

It is clear that that certain information (for example, Madzima: 76) persists in both repairs, since it does not participate in the violation of the functional dependency F1. On the other hand, some information (for example, Zanamwe: 50) does not persist in all repairs, because it participates in the violation of F1. There are other pieces of information that can be found in both repairs, for we know that there is a student with the name Zanamwe. Such information cannot be obtained if we simply discard the tuples participating in the violation.

In addition to the foregoing repairs, Libkin (n.d.) claims that database repairs depend on types of integrity constraints imposed on the database. For instance in the above example if constraints are functional dependences: say Name⟶ Mark, one of the tuples that violate the functional dependency must be deleted such that the repair will either be

STUDENT1

Name	Mark
Zanamwe	50
Madzima	76
Musara	67

But not both as is in the first case

OR

STUDENT2

Name	Mark
Zanamwe	75
Madzima	76
Musara	67

On the other hand, if constraints are referential integrity constraints, and you are given the referential integrity constraint, R[X] ⊆ S[Y] and the following relations:

R

X	**R**
x1	r1
x2	r2

S

Y	**Q**
x1	q1
x3	q2

To repair the above database which violates referential integrity constraints a tuple must be added to relation S such that it becomes:

S

X	**Q**
x1	q1
x3	q2
x2	q3

Another example to show how database repairs are done is as follows: let database instance D = {S (a), S (b), and Q (b)}. D is inconsistent with respect with IC: ∀x (S(x) ⊆ Q(x)). Consistency can be minimally restored by: Inserting Q(a): D1 = {S(a),Q(a), S(b),Q(b)} or by eliminating S(a): D2 = {S(b),Q(b)}. If we query S(x), the only consistent answer is tuple b

In addition to above assertions (by Libkin), Xie and Yang (2007) note that, if a database is inconsistent with respect to the key constraint two repairs are possible. For example, it is assumed that a relational schema R (reg#, *sex*), and database instance I = {(*r0020219*, *male*), (*r0020219*, *female*), (*r0020345*, *female*)}. I is inconsistent with respect to the key constrain. The two repairs are: $I1$ = {(*r0020219*, *male*), (*r0020345*, *female*)} and $I2$ = {(*r0020219*, *female*), (*r0020345*, *female*)}. It is clear that all the repairs have a minimal distance to the inconsistent database, but {(*r0020219*, *female*)} and {(*r0020345*, *female*)} are not repairs because their distances with respect to I are not minimal under set inclusion.

The minimality condition for the repairs is crucial in the definition. Otherwise, the empty set would trivially be a repair of every instance. For example, let $q1$ (*reg#*) = ∃ *sex*:*R*(*reg#*, *sex*). The consistent answers for $q1$ on I are (*r0020219*) and (*r0020345*). Let $q2$ (*reg#*, *sex*) = R (*reg#*, *sex*). The only consistent answer for $q2$ on I is (*r0020345*, *female*). Notice that the tuples (*r0020219*, *male*) and (*r0020219*, fe*male*) are not consistent answers. The reason is that neither of them is present in both repairs, which reflects the fact that *r0020219's* sexes are inconsistent.

It is evident from the above that, to repair a database that violates integrity constraints, either tuples have to be deleted or inserted. In the most cases, it could be difficult or undesirable to repair the database in order to restore consistency. This might be because, the process may be too expensive, useful data may be lost, one might not have permission to repair a database and restoring consistency can be a complex process. One strategy for managing inconsistent databases is to obtain consistent data without repairing the inconsistent database first. This technique is discussed below.

Consistent Answering Techniques Without Reparation

The approach is also known as, consistent query answering (CQA). This approach seeks to resolve inconsistencies at query time over inconsistent databases. Consistent query answering is the problem of retrieving "consistent" answers over inconsistent databases with respect to a set of integrity constraints (Xie and Yang, 2007). The approach involves a number sub-approaches namely, transformation of the query to obtain consistent answers, consistent query answer in the presence of inconsistent databases, to use bounded paraconsistent inference, detecting the cause of the inconsistency and retrieving a subset of the original database. The next section looks at some of the computational techniques for CQA

4.1 Consistent Query Answer in the presence of inconsistent databases

This technique is based on the postulation that an inconsistent database is not necessarily going to be repaired in a way that fully restores its consistency. This implies that, if such a database is to be queried, a distinction has to be made between the information in the database that violates integrity constraints, and one that does not. Typically, only a small part of a database will be inconsistent. It is therefore imperative to make precise the notion of "consistent" or "correct" information in an inconsistent database.

The following example presents the basic intuitions behind the notion of consistent query answer:

Consider a database subject to the following *Integrity Constraints*:

$$\forall x(R(x) \supset S(x)):$$

The instance $\{R\ (a),\ R\ (b),\ S\ (a),\ S(c)\}$ violates this integrity constraint. Now if the query asks for all x such that $S(x)$, only a is returned as an answer consistent with the integrity constraint. So this approach will only give answers that satisfy integrity constraints and ignore inconsistent information.

4.2 Transformation of queries to obtain consistent answers (Query rewriting)

This is a computational mechanism for retrieving consistent answers from databases that violate integrity constraints. Arenas et al (1999) note that, given a first-order query Q and an inconsistent database instance r, instead of explicitly computing all the repairs of r and querying all of them, a new query T(Q) is computed and posed to r the only available database. The answers to the new query are expected to be the consistent answers to Q. Xie and Yang (2007) assert that, the goal of query rewriting is to identify the subclasses of consistent queries to obtain consistent answers by rewriting the query into a new first-order query. The approach has two advantages namely: (a) allowing for consistent query answer in PTIME in data complexity; (b) the original query is rewritten into a new first-order query, which could be expressed in SQL. In such a case, consistent query answer can be done using the same query engine. On the other hand, this approach is only limited to the polynomial subclasses of the problem.

Arenas et al. (1999) first propose a method to compute consistent query answers based on query rewriting. The rewriting applies to and produces first-order queries, which draws on semantic query optimization techniques. A literal in the query can be resolved with an integrity constraint to form a residue. The method is an iterative operator that computes a sequence of queries. At each step of the iteration, each of the literals in the query gets its residues appended to it by means of a conjunction. Residues are associated with single literals $P(x)$ or $\neg P(x)$ (only one of each for every database relation p). Then all such residues are conjoined with the literal to form their expanded versions. If a literal that has been expanded appears in a residue, the residue has to be further expanded until no more changes occur. For each literal $P(x)$ (resp. $\neg P(x)$) and each constraint containing $\neg P(x)$ (resp. $P(x)$) in its clausal form (possibly after variable renaming), a local residue is obtained by removing $\neg P(x)$ (resp. $P(x)$) and the quantifiers for x from the (renamed) constraint.

Bertossi (2006) indicate that, query rewriting involves taking the original query Q that expects consistent answers, and syntactically transform it into a new query Q_0, such that the *rewritten query* Q_0, when posed to the original database, obtains as usual answers, the consistent answers to query Q. The essential question is, depending on the language in which Q is expressed, what kind of language is necessary for expressing the rewriting Q_0. The answer to this question should also depend on the kind of ICs being considered. It has to be noted that the new query collects as normal answers those tuples where the value of the first attribute is not associated to two different values of the second attribute in the relation. The diagram below clearly shows how query rewriting works in practice.

Using the above example, a query to select all students from the relation student such as

```
SELECT Name, Mark
FROM Student;
```

Will give an inconsistent answer, so the query can be rewritten as

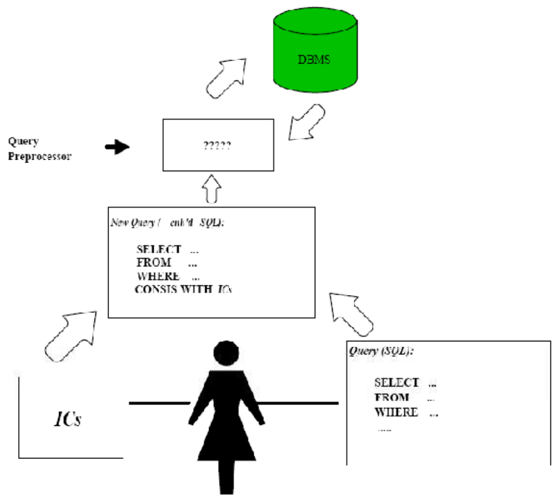
```
SELECT Name, Mark
FROM Student
CONSISTENT WITH
FD (Name;Mark);
```

Bertossi (2006) claims that query rewriting has the following drawbacks:

1. It can only be applied to certain SQL queries, essentially conjunctions of database tables.
2. Only applicable to certain integrity constraints, essentially universal integrity constraints. This covers most integrity constraints found in database praxis, exclusive of referential integrity constraints.
3. It only works with more expressive queries associated with referential integrity constraints

Similarly, Arenas et al (1999) comment that the query rewriting approach has some limitations, notably, the iterative operator associated with the approach only works for queries that are conjunctions of literals and universal integrity constraints but when applied to disjunctive or existential queries, completeness is lost

The diagram below shows how the query rewriting technique works.

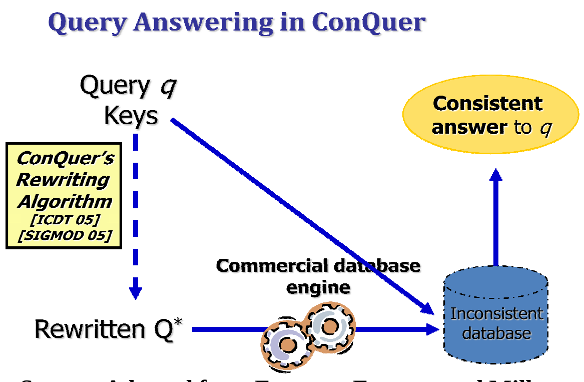


Source: adapted from Bertossi (2006)

4.3. ConQuer (Consistent Querying)

Fuxman, Fuxman and Miller (2005) present a version of query rewriting known as ConQuer. They note that ConQuer is a system for efficient and scalable answering of Structured Query Language (SQL) queries on databases that may violate a set of integrity constraints. Further they assert that, ConQuer permits users to postulate a set of key constraints together with their queries. The system rewrites the queries to retrieve all (and only) data that is consistent with respect to the constraints. The rewriting is into SQL, so the rewritten queries can be efficiently optimized and executed by commercial database systems.

Fuxman, Fuxman and Miller (2005) indicate that one of the practical applications of ConQuer is in Customer Relationship Management (CRM) where an integrated customer database is designed by drawing information from various autonomous sources such as sales, shipping, customer support, web forms and demographic data. Fuxman, Fuxman and Miller (2005) further postulate that, if data sources contain inconsistent data just transfer all the data to the integrated database and then rewrite your queries such that only consistent answers are obtained. The diagram below is a pictorial representation of how query answering in ConQuer operates.


Query Answering in ConQuer
Query q
Keys
ConQuer's Rewriting Algorithm [ICDT 05] [SIGMOD 05]
Consistent answer to q
Commercial database engine
Inconsistent database
Rewritten Q*

Source: Adapted from Fuxman, Fuxman and Miller (2005)

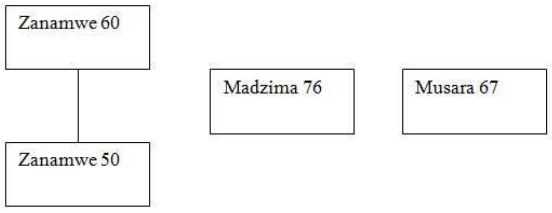
4.4. Conflict graphs

This approach bases a query answering process on the notion of conflict graph. The conflict graph constitutes a compact, space-efficient representation of all the repairs of a given database instance with respect to a set of integrity constraints. The repairs correspond to maximal independent sets of the graph. Chomicki, Marcinkowski and Staworko (2004) establish that this representation is specifically geared toward denial constraints. Chomicki et al. (2003) consider projection-free relational algebra queries with union and set difference, and extends the tractability of Consistent Query Answer (CQA) with respect to a set of denial constraints. With conflict graphs, the vertices of the graph are the tuples in the database; an edge connects two vertices if they violate together an integrity constraint. Given a relation student as shown below:

STUDENT

Name	Mark
Zanamwe	60
Zanamwe	50
Madzima	76
Musara	67

Two conflict tuples {(*Zanamwe*, 60), (*Zanamwe*, 50)} are the vertices of the conflict graph, and an edge connects the two vertices. Here follows the conflict graph.



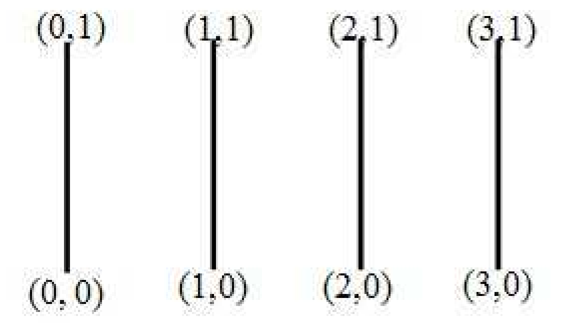
The conflict graph can be used to compute consistent answers to queries. Chomicki et al. (1999) describe a PTIME algorithm for CQA that is applicable to quantifier-free queries. The algorithm is also applicable to denial constraints that generalize functional dependencies. In this case, the notion of the conflict graph is replaced by that of the conflict hyper graph. The algorithm is as follows:

i. a conflict detection is performed to produce the conflict hyper graph over the database before processing the query;
ii. the original query is expressed in projection-free relational algebra, and it is translated into a quantifier-free first-order formula;
iii. the quantifier-free first-order formula is grounded with an appropriate set of bindings for its variables;
iv. the grounded formula and the conflict hyper graph are sent to the HProver PTIME algorithm, and thus evaluated in PTIME.

The entire process is implemented by the Hippo system, and the output of the system is the consistent answer to the input query. The above approach is practical even for large databases. However, the approach is still not applicable to queries with quantifiers or general universal constraints. Moreover, the number of conflicts in a database is not large, and the conflict graph does not require much space in the main memory.

Here follows another example of how conflict graphs are used to query inconsistent databases.

For any natural number *n* consider an instance
rn = {(0,0), (0,1), . . . , (*n*−1,0), (*n*−1,1)} of the schema *R*(*A*,*B*). Note that the set of all repairs of *rn* with respect to the functional dependency *A*→*B* is equal to the set {0, 1} *n* of all functions from {0 . . . *n*−1} to {0, 1}. It has already been indicated that, given a relation instance *r* and a set of functional dependencies *F*, a *conflict graph* is a graph whose vertices are the tuples of *r* and two tuples are adjacent only if they are conflicting with respect to a constraint from *F*. It has to be remembered that, conflict graphs are *compact representations of repairs* because the set of all repairs is equal to the set of all maximal sets of the corresponding conflict graph. The conflict graph for the instance *rn* for *n* = 4 and the functional dependency *A*→*B* is presented below.



4.5 Logic programs

Xie and Yang (2007) assert that a more general approach of retrieving consistent answers from inconsistent database is based on logic programs with stable model semantics. The approach can handle arbitrary relational calculus queries and binary universal constraints. More general queries could be considered, but ICs are restricted to be "binary", that is, universal with at most two database literals. There is a one-to-one correspondence between the database repairs and the stable models of the logic programs. The basic idea of the logic programs for CQA is as that we had better specify the class of repairs for reasoning with all the database repairs. From a logical specification of this class, different computations of consistent answers may be performed. This approach can handle all first-order queries and a much wider class of ICs than a query rewriting technique.

The major drawback of logic programs is that, since they work by grounding a logic program and use only main memory, these implementations can handle only relatively small databases. There are a number of systems for consistent query answering that rewrite queries into disjunctive logic programs. For instance, Informix focuses on expressiveness, more than efficiency and scalability. Such programs permit rewritings over general functional inclusion, and exclusion query constraints, but their overhead are more expensive for computation than SQL.

4.6 Preferred consistent query answers

Staworko, Chomicki, and Marcinkowski (n.d.) extend the framework of consistent query answers with an additional input consisting of preference information(φ). φ is used to define the set of preferred repairs Rep^{φ}. When consistent answers are computed, instead of considering the set of all repairs *Rep*, the set of preferred repairs is used. It is assumed that there exists a (possibly partial) operation of extending Φ with some additional preference information and we write $\Phi \subseteq \Psi$ when Ψ is an *extension* of Φ. Φ is considered to be *total* when it cannot be extended further. We identify the following desirable properties of families of preferred repairs:

1. **Non-emptiness:** $Rep^{\Phi} \neq \emptyset$ (P1)
2. **Monotonicity**: extending preferences can only narrow the set of preferred repairs
   $\Phi \subseteq \Psi \implies Rep^{\Psi} \subseteq Rep^{\Phi}$ (P2)
3. **Non-discrimination**: if no preference information is given, then no repair is removed from consideration $Rep^{\Phi} = Rep$. (P3)
4. **Categoricity**: given maximal preference information we obtain exactly one repair
   Φ is total$\Longrightarrow |Rep^{\Phi}| = 1$. (P4)

Here follows an illustration of the concept of preferred repairs. Assume you have a database consisting of the three binary relations: *Lectures (Tutor, Course), Department (Tutor, Department)* and *Course (Course, Department)* with the integrity constraint $\forall (T,C,D)[Lectures\ (T,C) \wedge Department\ (T,D) \supset Course\ (C,D)$ stating that if a tutor T lectures a course C and T is in the department D, then the course C must belong to the department D. Assume there are two different sources of the databases: D_1 = {Lectures (t_1, c_1), Lectures (t_2, c2), Department(t_1, d_1), Course (t_1, d_1) } and D_2 = Lectures(t_1, c_1), Department(t_2, d_1), Course (t_2,d_2). The two instances satisfy the constraint, but from their union we get a relation which does not satisfy the constraint. The presence of inconsistent data can be resolved by "repairing" the database. Informally, a repair for a possibly inconsistent database is a minimal set of insert and delete operations that make the database consistent, whereas a consistent answer is a set of tuples derived from the database, satisfying all integrity constraints.

Thus the integration of, possibly inconsistent, databases must consider the possibility of constructing an integrated consistent database by replacing inconsistent tuples. For instance, for the integrated relation of the above example, it is possible to obtain a consistent database by i) deleting the tuple Department (t_2, d_1), ii) deleting the tuple Lectures (t_2, c2), or iii) adding the tuple Course (t_2, d_1). These three update operations are repairs that make the database consistent, but one should prefer a repair with respect to an alternative one. For instance, one could prefer a repair which minimize the number of deletion and insertion of tuples in the relation Letcures and, in such a case, the first and third repairs are preferred to the second one, or one should prefer repairs minimizing the set of deletions and in such a case the third repair is preferred to the first two repairs.

Data Cleaning

Dasu and Johnson (2003) advocate that, data cleaning is one of the strategies for managing inconsistent databases. Similar sentiments are echoed by Lomet (2000); he says one way to deal with the impact of inconsistencies in the results of the query evaluation is data cleaning. Further, Dasu and Johnson (2003) note that data cleaning seeks to identify and correct data errors. However, the technique is semi-automatic and infeasible for some applications, for example, a user may want to adopt different cleaning strategies or retain all inconsistent data.

Dasu and Johnson (2003) also note that, the trend toward autonomous computing is making the need to manage inconsistent data more acute. There are an increasing number of applications whose data must be used with a set of independent constraints. Thus, a static approach with respect to a fixed set of constraints enforced by data cleaning may not be appropriate take for instance query rewriting. On the whole, despite the above limitations associated with this strategy, it is still useful and applicable in some scenarios when dealing with inconsistent databases.
Staworko, Chomicki, and Marcinkowski (n.d.) establish that the data cleaning system provides valuable information which may include:

– the timestamp of creation/last modification of the tuple (the conflicts can be resolved by removing from consideration old, outdated tuples),

– source of the information of the tuple (a user can consider the data from one source more reliable than the data from the other).

Staworko, Chomicki, and Marcinkowski (n.d.) also note that the approach of data cleaning has several shortcomings:

– If the user provides insufficient information to resolve all the conflicts then data cleaning results in an inconsistent database; this again may lead to misleading answers.

– Physically removing the tuples from the database may lead to information loss.

– Data cleaning does not allow using the incomplete information often expressed in inconsistencies.

References

[1] Arenas, M., Bertossi L, and Kifer M. (2000) Applications of annotated predicate calculus to querying inconsistent databases. In: Proceedings of the International Conference on Computational Logic. Berlin: Springer, 926–941

[2] Arenas, M., Bertossi, L. and Chomicki, J. (1999) Consistent Query Answers in Inconsistent Databases. In Proc. ACM Symposium on Principles of Database Systems (PODS'99). ACM Press, pp. 68-79.

[3] Bertossi, L. (2006) Consistent Query Answering in Databases. ACM Sigmod Record, 35(2):68-76.

[4] Bertossi, L. and Chomicki, J. (2003) Query Answering in Inconsistent Databases. In Logics for Emerging Applications of Databases, Springer, pp. 43-83.

[5] Caniup´an, M. and Bertossi, L. (2007) Optimizing and Implementing Repair Programs for Consistent Query Answering in Databases, Sistemas de Informaci´on Universidad del B´ıo-B´ıo Concepci´on, Chile

[6] Chomicki, J. (2007) Consistent Query Answering: Five Easy Pieces. In Proc. International Conference on Database Theory (ICDT'07), Springer LNCS 4353, pp. 1-17.

[7] Chomicki, J. and Marcinkowski, J. (2000) On the Computational Complexity of Consistent Query Answers, Bell Hall, Univ. at Buffalo, Buffalo, NY

[8] Dasu, T. and Johnson T. (2003) Exploratory Data Mining and Data Cleaning. New York: John Wiley.

[9] Fuxman, A., Fuxman, D. and Miller, R.J. (2005) ConQuer : A System for Efficient Querying Over Inconsistent Databases, Proceedings of the 31st VLDB Conference, Trondheim, Norway

[10] Libkin, L. (n.d.) Data Integration and Exchange

[11] Rahm, E. and Do, H.H. (2000) Data Cleaning: Problems and Current Approaches. IEEE Data Eng. Bull., 23(4):3–13,

[12] Staworko, S., Chomicki, J. and Marcinkowski, J. (n.d.) Preference-Driven Querying of Inconsistent Relational Databases,

[13] Vassiliadis, P., Vagena, Z., Skiadopoulos, S. and Karayannidis, N. (2000) ARKTOS: A Tool For Data Cleaning and Transformation

in DataWarehouse Environments. IEEE Data Eng. Bull., 23(4):42–47