ConQuer : A System for Efficient Querying Over Inconsistent Databases

Ariel Fuxman[†] Diego Fuxman[‡] Renée J. Miller[†]

[†]University of Toronto
Toronto, Canada

[‡]Universidad Nacional del Sur
Bahía Blanca, Argentina

Abstract

Although integrity constraints have long been used to maintain data consistency, there are situations in which they may not be enforced or satisfied. In this demo, we showcase ConQuer, a system for efficient and scalable answering of SQL queries on databases that may violate a set of constraints. 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.

1 Introduction

Integrity constraints have long been used to maintain data consistency. Data design focuses on developing a set of constraints to ensure that every possible database reflects a valid, consistent state of the world. However, integrity constraints may not be enforced or satisfied for a number of reasons. In some environments, checking the consistency of constraints may be too expensive, particularly for workloads with high update rates. Hence, the database may become inconsistent with respect to the (unenforced) integrity constraints. When data is integrated from multiple sources, each source may satisfy a constraint (for example a key constraint), but the merged data may not (if the same key value exists in multiple sources). More generally, when data is exchanged between independently designed sources with different constraints, the exchanged data may not satisfy the constraints of the destination schema.

One strategy for managing inconsistent databases is data cleaning [4]. Data cleaning techniques seek to identify and correct errors in the data and can be used to restore the database to a consistent state. Data cleaning, when applicable, may be very successful. However, these techniques are semi-automatic at best, and they can be infeasible or unaffordable for some applications. Furthermore, committing to a single cleaning strategy may not be appropriate for some environments. A user may wish to experiment with different cleaning strategies, or may desire to retain all data, even inconsistent data, for tasks such as lineage tracing. Finally, data cleaning is only applicable to data that contains errors. However, the violation of a constraint may also indicate that the data contains exceptions, that is, clean data that simply does not satisfy a constraint.

In our work, we will showcase the ConQuer system, which takes an approach that is applicable to databases with both errors and exceptions.[1] In ConQuer, a user may postulate a set of integrity constraints, possibly at query time, and retrieve all (and only) query answers that are consistent with respect to these constraints. Given a SQL query, ConQuer produces another SQL query that retrieves the consistent answers. Furthermore, ConQuer supports the detection and resolution of inconsistencies in the database.

2 Conquer's Approach

2.1 Examples

We will now illustrate ConQuer's rewriting strategy with some simple examples. The examples use the database of a movie theater (Figure 1). The database contains a relation `rooms`, with the rooms of the theater and the movies it is currently showing, and a relation `movies` with information about movies. We will assume that the key of `rooms` is the attribute `roomNo`, and that the key of `movies` is `movieName`.

[1] ConQuer stands for Consistent Querying

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

rooms

	roomNo	movieName
s1	#1	Les Invasions Barbares
s2	#2	Sideways
s3	#2	Million Dollar Baby
s4	#3	Madagascar
s5	#3	Sideways

movies

	movieName	country	year
t1	Million Dollar Baby	US	2004
t2	Million Dollar Baby	US	2005
t3	Sideways	US	2004
t4	Les Invasions Barbares	US	2003
t5	Les Invasions Barbares	Canada	2004

Figure 1: An inconsistent database with `rooms` and `movies` relations

Notice that both relations are inconsistent with respect to their keys. According to this database, the movie *Million Dollar Baby* has been released in two different years: 2004 (the correct one) and 2005. For the movie *Les Invasions Barbares*, the database gives two release years and two different countries: *US* and *Canada* (the correct one). In the `rooms` relation, rooms 2 and 3 violate the key constraint.

Suppose that a user wants a list of the movies that were released during or after 2004.

```
q1: select movieName
    from movies
    where year >= 2004
```

The answer to this query on the dirty database is {*Million Dollar Baby*, *Million Dollar Baby*, *Sideways*, *Les Invasions Barbares*}. The movie *Million Dollar Baby* is included in the answer twice, a fact which does tell the user that the database does not satisfy the key constraint. Furthermore, the query will retrieve all three movies, giving the user no indication that *Les Invasions Barbares* was included in the answer just because the underlying database is dirty or inconsistent. Similarly, the query answer does not indicate whether every (inconsistent) tuple for *Million Dollar Baby* is associated to a year greater than or equal to 2004. In contrast, ConQuer gives users the option of asking for query answers that are known to be consistent with respect to a set of constraints, in this case the key constraint of `movies`. In our example, ConQuer would rewrite query q_1 into the query given in Figure 2.

A key value is consistent if all possible tuples containing that key value satisfy the query. To understand the rewriting, notice that the first three lines correspond to the original query q_1. This part of the rewriting retrieves a set of answers that are candidates

```
select distinct m1.movieName
from movies m1
where m1.year >= 2004
      and not exists
          (select *
           from movies m2
           where m2.movieName=m1.movieName
                 and m2.year < 2004)
```

Figure 2: Rewriting of q_1

to be consistent answers. From the candidates, the rewriting filters out those key values for which there is another (possible) tuple that violates the query (the filter is implemented here in the nested `not exists` clause). In this example, the result of applying the rewritten query is {*Million Dollar Baby*, *Sideways*}. The movie *Million Dollar Baby* is included because while its data is inconsistent, all its tuples are associated to a year greater than or equal to 2004. The movie *Sideways* is in the consistent answer because it appears in a single, consistent tuple where the release year is 2004. However, *Les Invasions Barbares* is not returned since one of its tuples is for a year previous to 2004. This matches the intuition that we do not know *for sure* whether the movie was released in or after 2004. In addition to the nested `not exists`, we use the `distinct` keyword to ensure that each consistent answer is returned the right number of times (for this query, each consistent movie should be returned once).

The above example gives a flavor for the type of rewriting we will use to permit consistent query answering over a potentially inconsistent database. In ConQuer, we use a common definition of consistent query answer based on the notion of a repair [2]. For keys, a repair is a subset of the inconsistent database containing exactly one tuple per key. A repair is one possible "cleaned" version of the database. A query answer is said to be consistent if it is an answer to the query in every repair [2].

As another example, consider a query q_2, which retrieves the rooms of the theater showing movies released in or after 2004.

```
q2: select r.roomNo
    from movies m, rooms r
    where m.year>=2004
          and m.movieName=r.movieName
```

The consistent answer for q_2 on the database is room 2. The reason that room 1 is not a consistent answer is that it is showing the movie *Les Invasions Barbares*, which is not known (for certain) to have been released in or after 2004. Room 3 is not a consistent answer because it may be showing *Madagascar* or *Sideways*, the former of which is not a tuple in the `movies` relation.

```
with Candidates as (
   select distinct r.roomNo
   from movies m, rooms r
   where m.year>=2004
         and m.movieName=r.movieName),
Filter as (
   select roomNo
   from Candidates Cand
        join rooms r on Cand.roomNo=r.roomNo
        left outer join movies m
                on m.movieName=r.movieName
   where m.movieName is null or m.year<2004)
select roomNo
from Candidates Cand
where not exists (select * from Filter F
             where Cand.roomNo = F.roomNo)
```

Figure 3: Rewriting of q_2

In Figure 3, we show a query rewriting that computes the consistent answers for q_2. Notice that there are two subqueries named `Candidates` and `Filter`. `Candidates` corresponds to the original query q_2, except that it uses the `distinct` keyword. In this case, the result of applying `Candidates` to the database retrieves all the rooms. The query `Filter` returns the rooms that should be filtered out from the result of `Candidates` because they are not consistent answers. In this case, `Filter` returns rooms 1 and 3. Room 1 is returned by `Filter` because tuple $s1$ joins with tuple $t4$, which corresponds to a movie released before 2004. Room 3 is in `Filter` because the movie *Madagascar* does not appear in the relation `movies`, and therefore tuple $s4$ violates the join condition of q_2. Notice that `Filter` computes a left-outer join between `rooms` and `movies`. Since tuple $s4$ does not join with any tuple of `customer`, room 3 appears together with a null value for attribute `m.movieName` in the left-outer join. Therefore, the condition `m.movieName is null` is satisfied, and room 3 is returned by the filter.

The rewriting finally retrieves the rooms in the result of `Candidates` that are not in `Filter` (i.e., they are not filtered out). In this case, it returns room 1, which is the consistent answer.

2.2 Query Rewriting

The query rewriting algorithm of ConQuer is described in full elsewhere [6]. The following are some of its main advantages.

- The approach is fully declarative because it is based on an algorithm that rewrites SQL queries into SQL queries. Since it requires no procedural pre- or post-processing of the queries, ConQuer can be applied to much larger databases than any of the existing systems for consistent query answering [3, 5].

- In addition to select-project-join queries, ConQuer's algorithm works also for queries with grouping and aggregation. We consider both bag and set semantics (e.g., queries using the `DISTINCT` keyword). These features are needed to enable practical use in analysis queries over an integrated and potentially inconsistent data warehouse.

- ConQuer uses two kinds of rewritings: one that works over the unchanged (inconsistent) database, and another that makes use of a database with annotations about constraint violations. When available, ConQuer exploits these annotations using query optimizations specific to the semantics of consistent query answering. These optimizations are highly effective and would not be found by a standard optimizer.

- The efficiency of the query rewriting approach of ConQuer has been experimentally validated [6]. This included a detailed performance study of ConQuer's rewriting algorithm over data and queries from the TPC-H decision support benchmark [1]. The results show that the overhead of the rewritten queries is not onerous. In particular, for most of the tested queries, ConQuer retrieves the consistent answers within twice the time required to obtain the answer for the original (non-rewritten) query.

Notice that our approach is actually orthogonal to any cleaning that may be done on the database, and can be used to facilitate cleaning. In the absence of user input to decide between alternative tuples, our approach provides the option of retaining all alternatives rather than having to remove (or ignore) the inconsistent data. Furthermore, our approach may be used interactively to permit a user to understand where the data is potentially inconsistent. In particular, ConQuer presents the user with the difference between the results to the original and rewritten queries. This difference reflects inconsistent data that may need to be cleaned. In our example, the user may be alerted about the fact that the movie *Les Invasions Barbares* appears in the answer to the original query, but not in the consistent answer. This may prompt him or her to correct the data and change the year of tuple t_5 to 2003. After this change, the movie *Les Invasions Barbares* will no longer appear as a candidate answer to the original query.

3 System Overview

ConQuer is implemented in Java and follows a modular architecture. It consists of the following components:

- **Query Rewriting Module:** It rewrites a given SQL query into another SQL query that computes the consistent answers. If the database is annotated, it also produces the optimized query rewriting that makes use of annotations.

- **Query Execution Engine:** Connects with DB2 through a JDBC connection. This component can be used to compute both the original and the rewritten queries. It also computes the difference between the results of the queries in order to show the user some potential inconsistencies in the database.

- **Conflict Resolution Module:** Provides a tracing facility to find the data that leads to differences between the answer to the original query and the consistent answer. This module also permits a user to update the database to correct errors.

- **User Interface:** Query results are displayed using a web-accessible interface that is implemented in PHP.

4 Demonstration

In the demo, we will showcase the ConQuer system on a database that integrates information from a variety of sources. The user will be presented with an extensive menu of queries highlighting the inconsistencies present in the data. Furthermore, users will be given the freedom to try their own queries.

Once a query is submitted, the user interface will show the rewriting that uses annotations and the one which does not. The user will be allowed to choose which query rewriting should be executed on the database. Once the query is executed, the system will present three windows showing the answer to the original query, the consistent answer, and the difference between the previous two. In the last window, it will be possible to click on a tuple and invoke the Conflict Resolution Module to trace the reasons for the difference between the answer to the original query and the consistent one.

References

[1] TPC Benchmark H (Decision Support). Standard Specification Revision 2.1.0, 2003.

[2] M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *PODS*, pages 68–79, 1999.

[3] J. Chomicki, J. Marcinkowski, and S. Staworko. Hippo: A System for Computing Consistent Answers to a Class of SQL Queries. In *EDBT*, pages 841–844, 2004. Demonstration paper.

[4] T. Dasu and T. Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley, 2003.

[5] T. Eiter, M. Fink, G. Greco, and D. Lembo. Efficient Evaluation of Logic Programs for Querying Data Integration Systems. In *ICLP*, pages 163–177, 2003.

[6] A. Fuxman, E. Fazli, and R. J. Miller. Efficient management of inconsistent databases. To appear at SIGMOD, 2005.