Hippo: a System for Computing Consistent Answers to a Class of SQL Queries

Jan Chomicki[1], Jerzy Marcinkowski[2], and Slawomir Staworko[1]

[1] Dept. Computer Science and Engineering, University at Buffalo
{chomicki,staworko}@cse.buffalo.edu
[2] Instytut Informatyki, Wroclaw Uniwersity, Poland
Jerzy.Marcinkowski@ii.uni.wroc.pl

1 Motivation and Introduction

Integrity constraints express important properties of data, but the task of preserving data consistency is becoming increasingly problematic with new database applications. For example, in the case of integration of several data sources, even if the sources are separately consistent, the integrated data can violate the integrity constraints. The traditional approach, removing the conflicting data, is not a good option because the sources can be autonomous. Another scenario is a long-running activity where consistency can be violated only temporarily and future updates will restore it. Finally, data consistency may be neglected because of efficiency or other reasons.

In [1] Arenas, Bertossi, and Chomicki have proposed a theoretical framework for querying inconsistent databases. *Consistent* query answers are defined to be those query answers that are true in every repair of a given database instance. A *repair* is a consistent database instance obtained by changing the given instance using a minimal set of insertions/deletions. Intuitively, consistent query answers are independent of the way the inconsistencies in the data would be resolved.

Example 1. Assume that an instance of the relation *Student* is as follows:

Name	*Address*
Smith	*Los Angeles*
Smith	*New York*
Jones	*Chicago*

Assume also that the functional dependency *Name* → *Address* is given. The above instance has two repairs: one obtained by deleting the first tuple, the other - by deleting the second tuple. A query asking for the address of *Jones* returns *Chicago* as a consistent answer because the third tuple is in both repairs. However, the query asking for the address of *Smith* has no consistent answers because the addresses in different repairs are different. On the other hand, the query asking for those people who live in *Los Angeles* or *New York* returns *Smith* as a consistent answer.

This conservative definition of consistent answers has one shortcoming: the number of repairs. Even for a single functional dependency, the number of repairs can be exponential in the number of tuples in the database [3]. Nevertheless, several practical mechanisms for the computation of consistent query answers *without computing all repairs* have been developed (see [5] for a survey): query rewriting [1], logic programs [2, 4, 9], and compact representations of repairs [6, 7]. The first is based on rewriting the input query Q into a query Q' such that the evaluation of Q' returns the set of consistent answers to Q. This method works only for SJD[3] queries in the presence of universal binary constraints. The second approach uses disjunctive logic programs to specify all repairs, and then with the help of a disjunctive LP system [8] finds the consistent answers to a given query. Although this approach is applicable to very general queries in the presence of universal constraints, the complexity of evaluating disjunctive logic programs makes this method impractical for large databases.

[3] When describing a query class, P stands for projection, S for selection, U for union, J for cartesian product, and D for difference.

2 The System Hippo

The system Hippo is an implementation of the third approach. All information about integrity violations is stored in a *conflict hypergraph*. Every hyperedge connects the tuples violating together an integrity constraint.

Using the conflict hypergraph, we can find if a given tuple belongs to the set of consistent answers without constructing all repairs [6]. Because the conflict hypergraph has polynomial size, this method has polynomial data complexity and allows us to efficiently deal even with large databases [7]. Currently, our application computes consistent answers to SJUD queries in the presence of denial constraints (a class containing functional dependency constraints and exclusion constraints). Allowing union in the query language is crucial for being able to extract indefinite disjunctive information from an inconsistent database (see Example 1).

Future work includes the support for restricted foreign key constraints, universal tuple-generating dependencies and full PSJUD[4] queries. However, because computing consistent query answers for SPJ queries is co-NP-data-complete [3, 6], polynomial data complexity cannot be guaranteed once projection is allowed.

The whole system is implemented in Java as an RDBMS frontend. Hippo works with any RDBMS that can execute SQL queries, and provides a JDBC access interface (we use PostgreSQL). The data stored in the RDBMS needs not be altered.

The flow of data in Hippo is presented on Figure 1. Before processing any input query, the system performs *Conflict Detection* and creates *Conflict Hypergraph* for further usage. We are assuming that the number of conflicts is small enough for the hypergraph to be stored in main memory. The only output of this system is the *Answer Set* consisting of the consistent answers to the input *Query* in the database instance *DB* with respect to a set of integrity constraints *IC*.

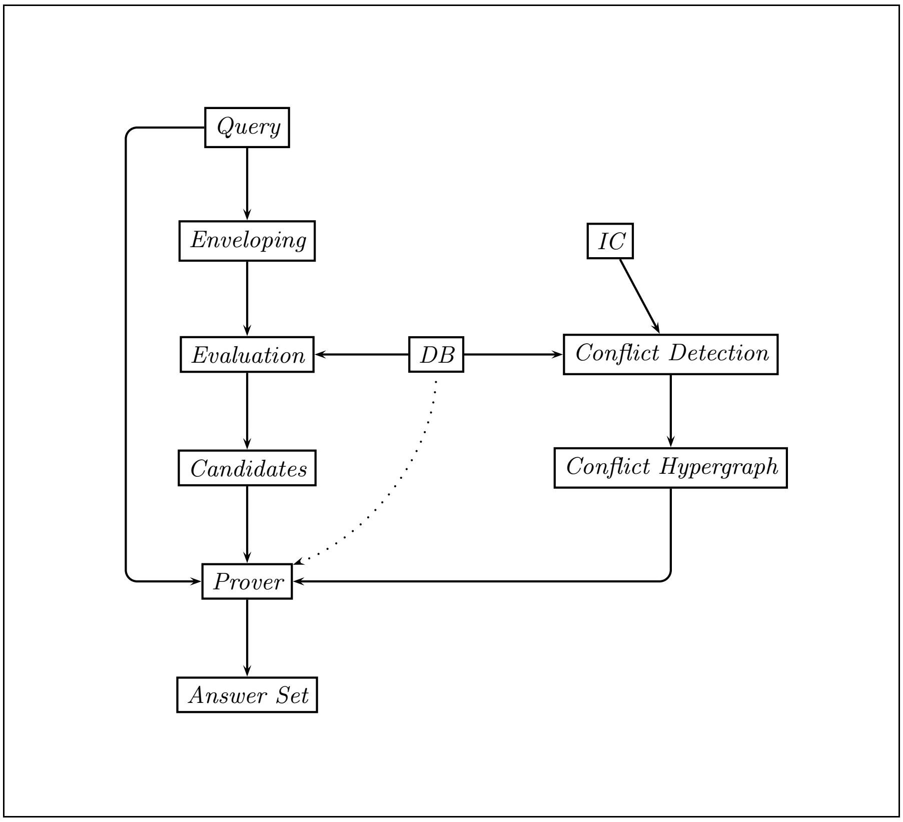

Fig. 1. Data flow in Hippo

[4] Currently, our application supports only those cases of projection that don't introduce existential quantifiers in the corresponding relational calculus query.

The processing of the *Query* starts from *Enveloping*. As a result of this step we get a query defining *Candidates* (candidate consistent query answers). This query subsequently undergoes *Evaluation* by the RDBMS. For every tuple from the set of candidates, the system uses *Prover* to check if the tuple is a consistent answer to the *Query*. Depending on the result of this check, the tuple is either added to the *Answer Set* or not.

For every tuple that *Prover* processes, several membership checks have typically to be performed. In the base version of the system this is done by simply executing the appropriate membership queries on the database. This is a costly procedure and it has a significant influence on the overall time performance of the system. We have introduced several optimizations addressing this problem. In general, by modifying the expression defining the envelope (the set of candidates) the optimizations allow us to answer the required membership checks without executing any queries on the database. Also, using an expression selecting a subset of the set of consistent query answers, we can significantly reduce the number of tuples that have to be processed by *Prover*. A more detailed description of those techniques can be found in [7].

3 Demonstration

The presentation of the Hippo system will consist of three parts. First, we will demonstrate that using consistent query answers we can extract more information from an inconsistent database than in the approach where the input query is evaluated over the database from which the conflicting tuples have been removed. Secondly, we will show the advantages of our method over competing approaches by demonstrating the expressive power of supported queries and integrity constraints. And finally, we will compare the running times of our approach and the query rewriting approach, showing that our approach is more efficient. For every query being tested, we will also measure the execution time of this query by the RDBMS backend (it corresponds to the approach when we ignore the fact that the database is inconsistent). This will allow us to conclude that the time overhead of our approach is acceptable.

References

1. M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 68–79, 1999.
2. M. Arenas, L. Bertossi, and J. Chomicki. Answer Sets for Consistent Query Answering in Inconsistent Databases. *Theory and Practice of Logic Programming*, 3(4–5):393–424, 2003.
3. M. Arenas, L. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. Spinrad. Scalar Aggregation in Inconsistent Databases. *Theoretical Computer Science*, 296(3):405–434, 2003.
4. P. Barcelo and L. Bertossi. Logic Programs for Querying Inconsistent Databases. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 208–222. Springer-Verlag, LNCS 2562, 2003.
5. L. Bertossi and J. Chomicki. Query Answering in Inconsistent Databases. In J. Chomicki, R. van der Meyden, and G. Saake, editors, *Logics for Emerging Applications of Databases*. Springer-Verlag, 2003.
6. J. Chomicki and J. Marcinkowski. Minimal-Change Integrity Maintenance Using Tuple Deletions. Technical Report cs.DB/0212004, arXiv.org e-Print archive, December 2002. Under journal submission.
7. J. Chomicki, J. Marcinkowski, and S. Staworko. Computing Consistent Query Answers Using Conflict Hypergraphs. In preparation.
8. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving in DLV. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer, 2000.
9. G. Greco, S. Greco, and E. Zumpano. A Logical Framework for Querying and Repairing Inconsistent Databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(6):1389–1408, 2003.