Core Schema Mappings: Computing Core Solution with Target Dependencies in Data Exchange

S. Ravichandra, and D.V.L.N. Somayajulu

***Abstract*—**Schema mapping is a declarative specification of the relationship between source schema and target schema. Nowadays, schema mapping is widely used in data transformation, schema evolution, data exchange and data integration activities and there by supporting in ETL applications, Enterprise Information Integration, and Enterprise Application Integration tasks. While performing data exchange for given schema mapping, it is required to consider all the types of target constraints which are target **tgds** (tuple generating dependencies) and target **egds** (equality generating dependencies). Many researchers have addressed various types of target constraints and provided solutions for them. The common drawback is that they are time consuming due to recursive nature and multiple intermediate states generation. In this paper, we proposed solution to resolve the above so called drawbacks and an approach to handle the target constraints on performing data exchange for given schema mapping in non recursive way. Our proposed solution rewrites given target dependencies and combines these with rewritten s-t tgds such that these modified dependencies can be directly translated into SQL Script. Later this produced SQL Script can be executed directly on a relational database system to populate data into target database

***Keywords*—**Schema mappings, core computation with target dependencies, handling target dependencies in data exchange, rewriting target dependencies.

I. Introduction

SCHEMA evolution is used widely to modify various changes at conceptual level such that these changes automatically reflect at logical and physical levels respectively. Different phases in Schema evolution are as follows: (i) Modification at conceptual level, (ii) Creation of schema, (iii) schema mapping from source schema to target schema and (iv) Data exchange. Algorithms were proposed by various authors to solve the problems related to schema evolution however there wide scope for improvement in terms of time and space optimization. Consider a typical relational database application development which has a schema designing phase in which the database schema is required to be evolved in subsequent versions. Each subsequent version of database schema is designed by transforming the current version of data in database schema.

Mr S.Ravichandra at present working as Assistant professor in Department of Computer Science and Engineering, National Institute of Technology Warangal, Telangana State, INDIA

D.V.L.N.Somayajulu at present working as Professor of Computer Science and Engineering at NIT, Warangal.

However this transformation was done manually by writing procedures which then gets executed directly on database system to produce instance of target schema. As the time required is more for this transformation so, it is required an automatic and efficient transformation mechanism from source schema to target schema.. In this paper, it is proposed to have an efficient non-recursive approach for transforming source schema to target schema under constraints.

Consider a scenario, where a source schema **S** and a target schema **T** are given. Schema mapping over **S** and **T** can be defined as relationship between **S** and **T** and more formally, the Schema mapping can be defined as three tuple: **M = (S,T,Σ)** Where **S** is source schema, **T** is target schema and **Σ** is high level declarative specification that specify relationship between **S** and **T**. Schema mappings studied in form of First Order (FO) logic. Basically, schema mappings can be divided in three forms namely **s-t tgds**, **target tgds** and **target egds**. Out of these s-t tgds is defined over **S** and **T** and rest both are defined over **T** itself. These all type of dependencies is given in [2] and data exchange can be described as: Data exchange via schema mapping **M = (S,T, Σ)** is transforming a given source instance **I** to a target instance **J**, such that **<I,J >** satisfies the dependencies **S** of **M**.

While performing data exchange, it is necessary to consider all type of dependencies. There are two methods which considered s-t tgds and rewritten dependencies such that these can be directly converted into executable SQL script. One of them is Laconic schema mappings [1] and another one is Core schema mappings: Scalable core computations in data exchange. But both failed to handle target dependencies. To overcome this issue, we give a method which we discuss in later sections. In a typical architecture of data exchange, a most general solution is called Universal solution. It is isomorphic to all other solution which satisfies schema mappings. But it suffers from inaccuracy and redundancy which results into following consequences:

1. Data increases exponentially when further schema evolution is performed.
2. Since instance of target schema is redundant so it causes problem in handling of the arbitrary query to the instance of target schema.
3. Query results could be redundant due to redundancy present in instance of target schema.

So one more solution was introduced i,e Core solution. Core solution is smallest and unique up to isomorphism. Both methods mentioned above find core solution by taking only s-t tgds under consideration. But not considered target

dependencies. So many problems occur like no accurate answers of conjunctive queries fired on target database and incomplete data present on target database. That's why, it is necessary to consider all type of dependencies.

A. Earlier work on core computation

The notion of core universal solution was first introduced in [5]. First polynomial time algorithm for computing was given in [5]. It was greedy [5] algorithm for computing core of universal solution. And author again modified greedy algorithm and given blocks [5] algorithm. But both algorithms were recursive and time consuming. Later, researchers started computing core by rewriting dependencies such that these dependencies can be directly converted into SQL script. The most efficient algorithms based on rewriting are Laconic schema mappings: Computing the core with SQL queries and Core schema mappings: Scalable core computations in data exchange. These methods are very much efficient but some points are not considered. Out of which, most crucial problem is that target dependencies was not considered. **The paper is organized as follows**. Section 2 formally defines the problem, Issues related to target dependencies and our contribution are discussed in section 3, section 4 defines the algorithms, section 5 defines the detailed description of the algorithm, section 6 discusses the results and section 7 concludes the work.

II. Problem Definition

Assume that there is a source schema S with data, an empty target schema T and schema mappings defined over $\mathbf{S \cup T}$. Basically, these schema mappings are categorized into two main types namely source-to-target dependencies and target dependencies. Since there are many algorithms which can generate core schema mappings for s-t tgds but none of them were able to handle the target dependencies [3]. In this paper, a mechanism is designed and implemented for transforming the data from source schema to target schema by ensuring the schema mappings $\mathbf{M = (S,T,\Sigma_{st}, \Sigma_t)}$ where $\mathbf{\Sigma_{st}}$ is s-t tgds and $\mathbf{\Sigma_t}$ is target tgds.

III. Issues Related To Target Dependencies

Following subsections discusses the most crucial issues related to target dependencies:

A.Finding transitive closure in target dependencies

Computation of core for schema mappings with target dependencies needs transitive closure to be computed. However it is not possible to define a small transitive relation in First Order (FO) logic. One simple example of transitive relation is T_1 (w, x, y, z) → T_2 (x, y, z) and T_2 (x, y, z) → T_3 (x, y) ∧ T_4 (x, z)

B. Cyclic Dependencies

It is hard to handle cyclic dependencies because it is required to decide which table gets populated with data as each table is derived from other table which is a cyclic dependent on the same table. One example of cyclic dependency is T_1 (x, y) → T_2 (x, y), T2 (x, y) → T_3 (y, x) and T_3 (x, y) → T_1 (x, y). But in case of weakly cyclic dependency, handling is easy. An example of weakly cyclic dependencies is T (x, y) ∧ T (y, z) → T (x, z). This cyclic dependency is easy to handle because dependency is applied when data is present in this table itself or already came from any other table

C. Order of execution of dependencies

As it is known that mapping is done target-to-target, so there could be scenario like figure 1. In this figure 1, table T gets derived from table A and B and further table T is used to populate data on table U. Conclusion is that table U cannot be populated before table T. Therefore, order of execution is necessary to decide.

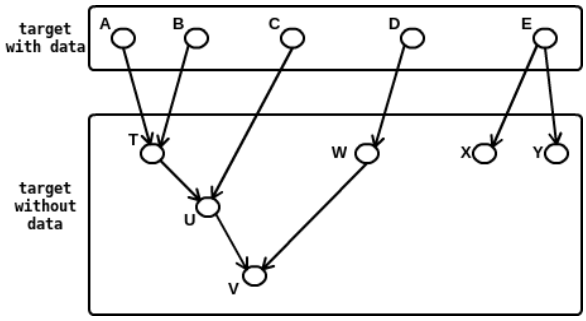


Fig. 1 Dependency graph

IV. Algorithm

In previous section, the limitations of other algorithms and issues related to target dependencies were discussed. The above mentioned limitations and issues (except cyclic dependencies) can be handled by our proposed algorithm.

Algorithm 1: Generate SQL script

Require: A schema mappings $M = (S, T, \mathbf{\Sigma_{st}}, \mathbf{\Sigma_t})$ where $\mathbf{\Sigma_{st}}$: Set of FO *s-t tgds*
$\mathbf{\Sigma_{st}}$: set of FO *target tgds* $\cup$ *target egds*
Ensure: *targetdependency* is acyclic.
1: Categorize dependencies into groups Σ_{g_i}
2: for all Σ_{g_i} *do* generate equivalent *JSON* dependencies
3: for all Σg_i
do
4: generate rewritten dependencies of E_{g_i} (refer algo2)
5: generate SQL queries from rewritten dependencies
6: add SQL queries into SQL script
7: end for

Output of this algorithm *i.e.,* SQL Script can be directly executed on database system to populate data into target database.

V. Detailed Working Of Algorithm

Firstly, We give overview of our algorithm and then we discuss step by step procedure of our algorithm in details. Our algorithm focuses on target dependencies and Laconic schema mappings [1] is used to rewrite *s-t tgds* followed by our algorithm to rewrite *target tgds*. This whole process results into SQL Script which is used to populate target database.

A. Overview of Algorithm

Our algorithm to compute core solution works in four phases (algorithm-I). For rewriting schema mappings, schema mappings are divided into two groups namely s-t dependencies and target dependencies. Then Laconic schema mapping is used to rewrite s-t dependencies and our algorithm rewrites target dependencies. Finally on putting together both, core schema mapping is formed that can be directly converted into SQL Script. Our algorithm works fine for all type of target dependencies except cyclic target dependencies (mentioned in III-B). Upcoming sections give detailed information of working of our algorithm.

B. Inputs to our Algorithm

Since our algorithm computes core solution $\mathbf{J_c}$ of source instance **I** for the given schema mapping M. So necessary inputs to the algorithm are following:

M: schema mapping which is $\Sigma_{st} \cup \Sigma_t$
S: source schema
T : target schema
I: instance of source schema

Schema mapping M must be in the form of FO (first order logic).

C. Data structures used

JSON [4] [8] format is used to handle dependencies. The main purpose of using JSON format to find transitive closure (mentioned in III-A) which is not possible in First Order (FO) logic. JSON also has some additional benefits like it has a fix layout and stores values in form of key-value pairs and almost all programming languages provides a very good API for accessing and modifying JSON Documents. First Order(FO) target dependency, $T_I (x, y, z) \wedge T_2 (x, u, v) \rightarrow T (x, y, u, v, w)$, JSON representation is as follows:

Conclusion representation:

target_ table:"T", target_ table_ attributes:["I" , "2" , "3" , "4" , "nm"]
Atoms of premise representation:
{"table":"T_I ","attributes":["I" , "2" , "nm"]},
{"table":"T_2 ","attributes":["I" , "3" , "4"]}
On putting conclusions, premises together, above dependency representation in JSON format is as follows:

{target_ table:"T", target_ table_ attributes:["1" , "2" , "3" , "4" , "nm"], source_tables:[{"table":"T_1","attributes":["1" , "2" , "nm"]},{"table":"T_2 ","attributes":["1" , "3" , "4"]}]}
Where numbers used will help us in mapping between source tables attributes to corresponding target tables attributes. And nm shows not mapped attributes.

Another data structure is disjoint-set data structure [7]. It is used to draw table relation in graph. It is used to predict order of execution of queries.

D. Categorizing dependencies into groups of same types

Schema mappings are divided into two groups namely s-t dependencies and target dependencies. Dependencies under s-t dependencies group have premise of conjunction of atoms of source schema and conclusion of conjunction of atoms of target schema. Dependencies under s-t dependencies group have both premise and conclusion of conjunction of atoms of target schema. After categorizing, these groups are handled in sequentially. Firstly, s-t dependencies is handled by Laconic schema mappings rewriting algorithm followed by handling of target dependencies using our algorithm. Following subsection shows detailed explanation of our algorithm only and only target dependencies are taken under consideration

E. Generating equivalent JSON dependencies

Target dependencies can be expressed in JSON format. In section V-C, conversion is shown. After conversion into JSON dependencies, rewriting process of these dependencies is performed.

F. Generating rewritten dependencies

Generating rewritten dependencies is three step process namely re-ordering to get phases, grouping dependencies with same conclusion to get sets and perform full outer join for each set based on common attributes. But before proceeding to rewriting process, it is necessary to check for cyclic dependencies as our algorithm does not work for cyclic dependencies

---

Algorithm 2 : Rewriting JSON dependencies and generate SQL script

---

Require: *JSON* dependencies and directed dependency graph of dependencies

1: Perform cyclic dependency check using DFS (depth first search) [unknown reference] in directed dependency graph.
2: Remove cyclic dependencies if exists.
3: Construct Phases (*P*) by level order traversal
4: for all $P_i \in P$ do
5: Construct Sets (*S*) by grouping dependencies with same conclusion
6: for all $S_j \in S$: $S_j = \{d_1, d_2, ..., d_n\}$ and
7: $d_k : premise_k \rightarrow conclusion_k$
8: do
9: Convert each premise of dependencies into SQL query
10: Perform full outer join based on common attributes of conclusion
11: $remise_1 \bowtie premise_2 \bowtie \ldots \bowtie premise_n$
12: where $\bowtie$ is full outer join
13: end for
14: end for

To perform cyclic check for dependency and re-ordering of dependencies, dependency graph for dependencies is needed.

Dependency graph for dependencies: each node represents atoms of premise or conclusion and edge from premise atom to related conclusion atom. JSON dependency from section V-C can be represented into dependency graph in figure 2:

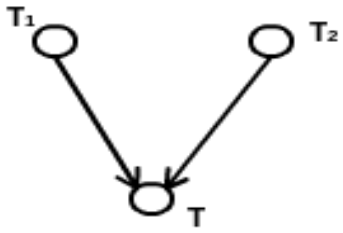


Fig. 2 Dependency graph for JSON dependency given in section V-C

Cyclic check: Check for cyclic dependencies can be performed using DFS (depth first search) [6]. DFS algorithm tells presence of cycle in directed dependency graph which means there is presence of cyclic dependency in target dependencies. After eliminating these cyclic dependencies our algorithm can be applied for rest target dependencies.

After checking for cyclic dependencies, rewriting process is started.

Constructing phases: Re-ordering is performed to construct Phases. For re-ordering, level order traversal is done. Level order traversal starts with zero level which consists of all atoms which are not the part of conclusion of target dependencies. Each level corresponds to a phase for example level i is phase i In figure 1 traversal is started with level 0 having atoms

All = {A,B,C,D,E }

Now, the very first phase is constructed by fully and direct dependent on All and the atoms of this phase is added to All. Now phase1 and All looks like:

phase1 = {T,W,X,Y }
All = {A,B,C,D,E,T,W,X,Y}

For next phase same procedure is applied. Finally, we get following phases:
P1 = {T,W,X,Y }
P2 = {U}
P3 = {V}

Constructing sets: Now, sets are constructed for each phase (Pi). Set is formed by grouping dependencies with same conclusion. One simple example of set:

[{"target_table_attributes":["1","2","nm","nm"],"target_table":
"target_schema.served","
source_tables":[{"attributes":["1","nm","nm"],"table":
"source_schema.route"},
{"attributes":["2","1"],"table": "source_schema.ap"}]},
{"target_table_attributes":["1","2","nm","nm"], "target_table":
"target_schema.served",
"source_tables":[{"attributes":["nm","1","nm"],"table":
"source_schema.route"},
{"attributes": ["2", "1"], "table": "source_schema.ap"}]}]

Finally, we get sets from each phases in a particular order which collectively forms rewritten dependencies. Now these rewritten dependencies are converted into SQL script

G. Generating SQL script from rewritten dependencies

Start rewriting each set into SQL query. First of all, each dependency should be converted into SQL query. Then perform full outer join on dependencies of the set based on common attributes present in conclusion. Above set is converted into SQL as follows:

```
INSERT INTO target_schema.served
((SELECT DISTINCT source_schema.route.source as a1,
source_schema.ap.country as a2, NULL, NULL FROM
source_schema.route, source_schema.ap WHERE source
_schema.route.source = source_schema.ap.city FULL
OUTER JOIN
SELECT DISTINCT source_schema.route.destination as a3,
source_schema.ap.country as a4, NULL, NULL FROM
source_schema.route, source_schema.ap WHERE source_
schema.route.destination = source_schema.ap.city ON a1=a3
and a2=a4) MINUS (SELECT DISTINCT * FROM
target_schema.served))
```

Produced SQL script consists of such SQL queries which can be directly executed on database system to populate data on target schema. This populated schema with data called core solution (Jc) and it is final solution (J) such that it satisfies given schema mappings. Next section shows results and analysis.

VI. Implementations And Results Analysis

In this section, we briefly define and discuss the implementation needed for proving the efficiency and benefits of proposed work over other existing algorithms. Here, first we give the details of implementation along with required tools and software's and finally performance evaluation is done on the basis of results.

A. Implementation details

For performance evaluation, we implement naive chase procedure [1] for computing Universal solution along with our proposed algorithm which computes Core Universal Solution. For making fair comparison, we use same setup for both the algorithm. We use three sample databases consisting of 3, 15 and 22 tables in target schema.

B. Results

We compare both algorithms on the basis of redundancy available in databases. We measure redundancy in terms of tuples present in the target database after running both algorithms. Results may vary on changing schema mapping because mapping is totally based on situation of change in database. Figure 3 shows Universal Vs Core : Percentage of size reduction. Another observation was made the database system which is feasible and can be used in applications. Table 1 shows time of generation of SQL script and execution time of SQL script on database system and figure 4 shows graphical time comparison.

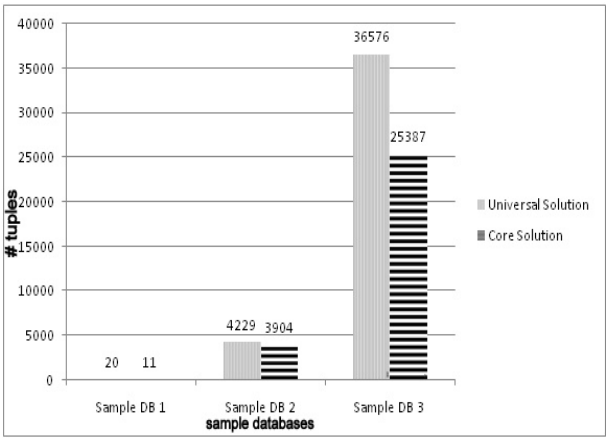
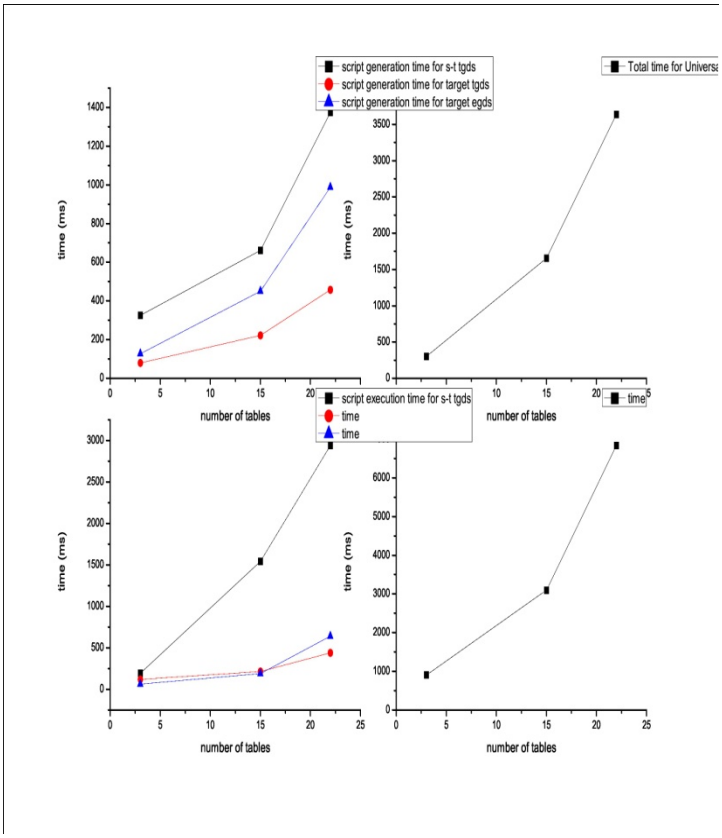


Fig. 3 Core vs Universal Solution: % size reduction

TABLE I
EXECUTION TIME OF SQL SCRIPT CONSTRUCTION AND EXECUTION ON SAMPLE DATABASES

Sample Databases	No of Tables	Execution time in milliseconds (Script construction time/script execution time)		
		s-t tgds	target tgds	target egds
Airports information	3	325/193	79/119	126/60
Employees	15	660/1543	222/213	450/187
Sakila	22	1373/2937	457/437	987/638



VII. SUMMARY AND CONCLUSION

We have analyzed the existing algorithms for computing universal and core solution and then proposed an approach by considering target dependencies with some restrictions as input. Our algorithm first generates core schema mappings and then it is converted automatically into SQL script. This SQL script is directly executed on database system.

Our approach is superior because it generates very few SQL queries for Σ_{st} (first order s-t schema mappings) and Σ_t (first order acyclic target dependencies) however the approach has a limitation to handle cyclic target dependencies.

REFERENCES

[1] Kolaitis B. ten Cate, L. Chiticariu and W. C. Tan. Laconic schema mappings: Computing the core with sql queries. pages 1006-1017, 2009.

[2] Ronald Fagin. Tuple-generating dependencies. http://www.springerreference.com/docs/html/chapterdbid/64189.html, June 2014. [Online; accessed 13-May-2014].

[3] Salvatore Raunich Giansalvatore Mecca, Paolo Papotti. Core schema mappings: Scalable core computations in data exchange. Pages 677-711, 2012.

[4] JSON. Introducing json. http://www.json.org/, June 2014. [Online;accessed 1-June-2014].

[5] P. G. Kolaitis R. Fagin and L. Popa. Data exchange: Getting to the core. *ACM TODS, 30(1)*, 30(1):174-210, 2005.

[6] Wikipedia. Depth-first search wikipedia, the ree Encyclopedia."http://en.wikipedia.org/w/index.php?title=Depth-first search& oldid=605994808, 2014. [Online; accessed 8-June-2014].

[7] Wikipedia. Disjoint-set data structure - Wikipedia, Encyclopedia.http://en.wikipedia.org/w/index.php?title=Disjoint-set dataucture&oldid=607693124, 2014. [Online; accessed 8-June-2014].

[8] Wikipedia. Json - Wikipedia, the free encyclopedia.http://en.wikipedia.org/w/index.php?title=JSON&oldid =611986835, 2014. [On- line; accessed 8-June-2014].

Mr S.Ravichandra at present working as Assistant professor in Department of Computer Science and Engineering, National Institute of Technology Warangal, Telangana State, INDIA. He is a Member of IEEE and ACM. His research interests are Software Engineering and schema evolution in Software engineering databases.

D.V.L.N.Somayajulu at present working as Professor of Computer Science and Engineering at NIT, Warangal. He is member and fellow of IETE, member of ACM, member of IEEE, Fellow of IE(I),and life member of ISTE. He has published around 35 research publications in various National and International conferences & Journals. He received Best Engineer of the Year award by IE(I) and APSCHE, Hyderabad in 2007. His interested research areas are incomplete databases, privacy, Big data and Software engineering.