AN APPROACH TO SCHEMA MAPPING GENERATION FOR DATA WAREHOUSING

The members of the Committee approve the masters thesis of Karthik Jagannathan

Sharma Chakravarthy _____
Supervising Professor

Mohan Kumar _____

David Kung _____

Copyright © by Karthik Jagannathan, 2002

All Rights Reserved

To My Wife, Family and Friends

AN APPROACH TO SCHEMA MAPPING GENERATION FOR DATA WAREHOUSING

by

KARTHIK JAGANNATHAN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2002

ACKNOWLEDGMENTS

Before anything, I would like to thank my advisor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on this challenging topic and for guiding and supporting me through the course of this research.

I would like to thank Dr. Mohan Kumar and Dr. David Kung for serving on my committee, and for their valuable comments and guidance.

This work was supported, in part, by the Office of Naval Research, the SPAWAR System Center-San Diego & by the Rome Laboratory (grant F30602-01-2-0543), and by NSF (grants IIS-0123730 and IIS-0097517).

I would also like to thank my wife for her endless love and constant support throughout my thesis; from the day I started to the day I defended my thesis.

November 18, 2002

ABSTRACT

AN APPROACH TO SCHEMA MAPPING GENERATION

FOR DATA WAREHOUSING

Publication No. _____

Karthik Jagannathan, M.S.

The University of Texas at Arlington, 2002

Supervising Professor: Sharma Chakravarthy

In data warehousing, the source schemas are defined independently from the warehouse schemas, which are typically designed based on the information need of the warehouse users. The mappings between the source and warehouse schemas are also determined manually. Typically, more than one mapping between the warehouse schema and the source schemas is possible and the designer might miss the most appropriate mapping from the viewpoint of updates and maintenance of the warehouse.

Automated generation of the mappings between the source and the warehouse schemas would generate a complete list of mappings from which the warehouse designer can choose the appropriate mapping.

The issues encountered during automation are numerous, including but not restricted to the presence of synonyms, homonyms and derived attributes in the source and warehouse schemas. This thesis focuses on automating mapping generation in data warehousing for the relational domain and handles select, project, join, union and intersection mappings.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

1.1 Overview

Time and again, the problem of "schema matching" keeps sprouting. Schema matching refers to the problem of finding mappings between the attributes of any given pair of schemas that are semantically related, or in other words, a homogeneous pair of schemas. Schema matching is a generic problem pertinent in many domains including XML, relational and object-oriented. Though the problem is comparable, the complexity of this problem is dependent on the richness of the schema involved.

To state an example, if one tried to map attributes between two XML schemas, one has to take the level in the hierarchy tree, and the sub-structure of the attributes involved. But if the same case were to be considered in the relational domain, attribute matching would involve matching attributes based on the name and the type.

1.2 Motivation

Schema matching plays an imperative role in data warehousing, wherein, given source and warehouse schemas, one is required to find the attribute mappings between the warehouse and the source schemas.

In a typical scenario, the source schemas are created independently and at various points in time. The warehouse schemas are designed based on the information need of the warehouse users/ analysts, and the warehouse designer determines the mappings between the warehouse and the source schemas manually.

Given source and warehouse schemas, one should note that there would be more than one possible mapping to generate a particular warehouse schema from the given source schemas. The warehouse designer might miss the most appropriate mapping from the viewpoint of updates and maintenance of the data warehouse as a result of trying to map the schemas manually.

1.3 Automating the process of mapping generation

Automating the process of the generation of the mappings between the source and the warehouse schemas would result in a system that requires less time and energy to be spent by the designer, and at the same time will be correct and complete. This would generate a complete list of all the mappings that are possible for the warehouse schemas from the source schemas, enabling the warehouse designer to choose the appropriate mapping for that given case. This would allow the designer to explore several mappings before finalizing the warehouse schemas.

1.4 Related work

Though not in the commercial world, where they still largely depend on manually generating the mappings, some research has been done in the area of schema matching. As the problem of schema matching is more generic in the sense that it is prevalent in many other domains including XML message mapping and schema integration [1], most of the research that has been done is largely domain specific i.e., pertaining to a specific domain or caters to a totally different set of requirements and needs. Thus the need for a domain specific solution for this generic problem arises.

1.5 Automation issues

Questions arise as to the simplistic straightforwardness in automating the process of schema mapping. But in reality, the issues encountered during automation are numerous. If the name space follows a single convention, then mapping would have been quite straightforward. But since the sources and hence the source schemas are created independently and at different points in time, one cannot assume a uniform naming convention for the attributes. Hence, the complications that arise are numerous and not limited to:

- Attributes that have the same name and same or different type, but need to be considered as two separate attributes (called *homonyms*)
- Attributes that are structurally similar but semantically dissimilar, that is, attributes with different names but which have the same type (called *synonyms*)
- An attribute of a warehouse schema that is referred to by a different name in the corresponding source schema
- Derived attributes of the warehouse schema, that are computed from more than one attribute of the source schemas (Cumulative GPA, Gross Total, etc.)

The process of automation should be able to comprehend and differentiate between these variations and arrive at a set of mapping that is relevant to the schemas in question. If these were not to be taken into consideration, it would result in an incomplete and incorrect generation of the mappings, as it would not have covered all cases. It is obvious that given the set of analogous, overlapping inputs, more than one plausible mapping can be arrived at. In the end, it is left to the warehouse designer to aptly choose the mapping appropriate to the case. Though validation by the designer still plays a major role, the objective is to generate

the set of mappings between the source and the warehouse schemas that are pertinent. Some of the pros and cons of the process of automation are summarized below:

Pros:

- Enables proper understanding of the mapping space
- Enables evaluation of mappings from different viewpoints
- Enables the warehouse designer to choose the appropriate schemas based on his understanding of the mapping space
- Eliminates the strenuous manual process and saves time
- Requires less exertion on the part of the designer as it generates all the mappings possible for the warehouse schemas
- Eliminates the possible fault on the part of the designer who might miss the most appropriate mapping from the viewpoint of updates and maintenance of the data warehouse

Cons:

- An initial effort on the designer in the form of the source file
- Designer has to validate the results and choose the appropriate mapping

From this, it is bare that it is still the designer who specifies the various semantics (synonyms, homonyms, and the like) that come into play in generating the mappings, as one requires a base on which to build the automation. Thus, automation here is not really complete, as some amount of interaction is involved at some level or the other. But this entirely eliminates any ambiguity or errors in the generated mapping that might otherwise occur as a result of human intervention, and generates all possible mappings that might be overlooked.

1.6 Problem statement

Given a set of semantically related source and warehouse schemas, the problem involves in automating the process of matching the schemas, and arriving at a resulting set of all possible mappings between the warehouse schemas and the given source schemas that are comparable. Once the appropriate mappings are determined, the problem also involves in generating the triggers and other code for propagation of the updates from the source to the warehouse schemas.

This forms the basis and the motivation for this thesis, which works towards an optimal solution for automating the process of schema matching and mapping generation, especially pertaining to data warehousing for the relational domain. The tool would enable understanding of the mapping space, and the evaluation of mappings from different viewpoints (ease of implementation, etc). But, it is up to the user to choose the most appropriate mapping (may be based on other considerations). The tool would also generate triggers and other code for propagation of the updates to the warehouse schemas. This thesis handles select-project-join (or SPJ), union and intersection mappings.

1.7 Contributions of this thesis

This thesis contributes to the following:

- ✓ Designing a solution for automating the process of schema matching and mapping generation for the warehouse schemas pertaining to data warehouses for the relational domain handling SELECT-PROJECT-JOIN, UNION and INTERSECTION mappings.
- ✓ Presenting the design of the algorithm and the implementation issues, including the handling of homonyms, synonyms and the attribute mappings

- ✓ Looking at performance issues and ways to improve the performance of the algorithm including parallelization

In the forthcoming chapters, the goal of automating schema matching and the generation of mappings is realized. Chapter 2 gives an overview of the related work in the area of automating schema matching. Chapter 3 gives a broad view of the design and the algorithm. Chapter 4 goes into the depths of the data structures involved in the design. Chapter 5 takes a walk through the various stages of implementation of the design and the implementation issues involved. Chapter 6 evaluates the performance of the algorithm and covers the various test cases that were tested out for consistency and correctness. Finally chapter 7 derives conclusions and discusses the scope for future work in this area, including parallelization leading to improvements in performance of the design.

CHAPTER 2
RELATED WORK

As mentioned in the previous chapter, some work has been done in this pervasive area of automating the process of schema matching. This chapter gives an overview of the existing research, and draws a parallel of the matching approach of this thesis with the rest, giving its pros and cons against the other systems.

2.1 Generic Vs domain-specific model

To start with, this thesis is working towards solving the problem of automating the process of schema matching and mapping generation specific to the relational domain, as we believe that at this point, it is more relevant to look at this problem from the perspective of relational systems, rather than trying to achieve a more generic system that would then have to be tweaked again for it to be of any use to the domain we want to use it under.

2.2 Overview of matching techniques

This section gives an overview of the different matching techniques that have been presented to date. One needs to note at this point that save a very few techniques, most of the rest have been/ are developed specific to a certain domain, for a definite cause. A list of the available techniques include - SemInt [2, 3], LSD [2, 4], SKAT [2, 5], TranScm [2, 6], DIKE [2, 7], ARTEMIS [2, 8] and CUPID [1]. Table 2.1 [2] gives an overview of the various matching techniques and their characteristics.

Table 2.1. Characteristics of matching techniques.

		SemInt	LSD	SKAT	TranScm	DIKE	ARTEMIS
Schema types		Relational, files	XML	XML, IDL, text	SGML, OO	ER	Relational, OO, ER
Metadata representation		Unspecified	XML, schema trees	Graph-based OO DB model	Labeled graph	Graph	Hybrid relational/ OO data model
Match granularity		Element-level:attributes	Element-level	Structure-level: classes	Element-level	Element/structure-level:entities/ relationships/ attributes	Element/structure-level:entities/ relationships/ attributes
Match cardinality		1:1	1:1	1:1 and 1:n	1:1	1:1	1:1
Schema-level match	Name-based	-	Name equality /synonyms	Name equality; Synonyms; Homonyms; Hypernyms	Name equality; Synonyms; Homonyms; Hypernyms	Name equality; Synonyms; Hypernyms	Name equality; Synonyms; Hypernyms
	Constraint-based	15 criteria: data type, length..	-	Is-a(inclusion); Relationship cardinalities	Is-a(inclusion); Relationship cardinalities	Domain compatibility	Domain compatibility
	Structure Matching	-	-	Similarity w.r.t. related elements	Similarity w.r.t. related elements	Matching of neighborhood	Matching of neighborhood
Reuse/auxillary information used		-	Comparison with training matches; lookup for valid domain values	Reuse of general matching rules	-	-	Thesauri
Combination of matchers		Hybrid matcher	Automatic; weighted combination of all learners per instance object; combination of instance predictions	-	Hybrid matchers; fixed order of matchers	-	Hybrid of name and structure matchers
Manual work/user input		Selection of match criteria	User-supplied matchers for training sources	Match/ mismatch rules	Resolving multiple matches, adding new matching rules	Resolving structural conflicts	User can adjust weights in match calculations and validate match choices
Application area		Data integration; 3 test cases	Data integration with pre-defined global schema	Ontology composition to support data integration/ interoperability	Data translation	Schema integration	Schema integration
remarks		Neural networks; c implementation		"algorithms" implicitly represented by rules	Rules implemented in Java	Algorithms to calculate new synonyms, homonyms, similarity metrics	Also embedded in the MOMIS mediator, with extensions

The following sections detail out the various systems and their relevance to this thesis, and the problem statement at hand.

2.2.1 SemInt

The SemInt match prototype [3] creates a mapping between individual attributes using neural networks to determine the same [2]. This does not support name-based matching.

2.2.2 LSD

The LSD (Learning Source Descriptions) system [4] uses machine-learning techniques to match a new data source against a previously determined global schema, producing a 1:1 atomic-level mapping [2]. This technique was developed mainly for the XML domain.

2.2.3 SKAT

The SKAT (Semantic Knowledge Articulation Tool) prototype [5] follows a rule-based approach to semi-automatically determine matches between two ontology [2]. This technique is relevant to the XML domain, and the schemas are transformed into a graph-based object-oriented database model.

2.2.4 TranScm

The TranScm prototype [6] uses schema matching to derive an automatic data translation between schema instances. Input schemas are transformed into labeled graphs, which is the internal schema representation [2]. This is relevant in the objected oriented domain.

2.2.5 DIKE

DIKE [7] focuses on finding pairs of objects in two schemas that are similar, in the sense that they have the same attributes and relationships, but are of different “types” [2]. The motivation involves in the need for schema abstraction for large systems. The solution

requires clustering objects into subsets and producing an abstracted schema obtained by substituting each subset with one single object representing that subset. The various steps involved are:

1. Enrichment of schema description obtained by semi-automatically extracting knowledge
2. Exploit inter-schema properties from the data repository
3. Exploitation of the repository derived in step 3 to support the designer in realizing a data warehouse over available data.

One requires to note here that this system is mainly used to support the designer of the warehouse by supplementing with additional information about the sources in question, which is very much different from the requirements of the problem statement as described in this thesis, which aims at solving the problem of generating the warehouse mappings, given the source and the warehouse schemas.

2.2.6 ARTEMIS

ARTEMIS [8] is a schema integration tool which completes schema integration by clustering attributes based on computed affinities and constructing views based on the clusters [2]. It is used to integrate independently developed schemata into a virtual global schema, the area of application of which is again differing from the one this thesis involves.

2.2.7 CUPID

CUPID [1] is a generic schema matching algorithm that discovers mappings between schema elements based on their names, data types, constraints and schema structure. The approach involves in attacking the problem by computing similarity coefficients between the elements of the two schemas and then deducing a mapping from those coefficients.

The input is via initial mapping, dictionary or thesaurus, library of known mapping etc. Calculating the coefficients are done in two stages namely, linguistic mapping and structural mapping. It involves normalization and clustering the schema elements into categories to reduce the number of element-element comparison. Comparison involves in calculating the linguistic similarity of each pair of elements from compatible categories, resulting in a table of linguistic similarity coefficients between elements of the comparing two schemas. The mapping generation involved in choosing pairs of schema elements with maximal weighted similarities.

Though the claim is that it is a generic schema matching algorithm, it is more specific to XML structures, and involves in finding similarities between schemas in a hierarchical manner, similar to XML. The algorithm leans towards a specific purpose in the XML domain.

2.3 Comparison

The following table [table 2.2] gives a comparison of the characteristics of the design approach of this thesis along with two other systems, the DIKE and the CUPID system that were somewhat comparable to what has been done in this thesis.

Table 2.2. Comparison of characteristics.

		DIKE	CUPID	OURS
Schema Types		ER	XML	Relational
Metadata Representation		Graph	Tree Graph	Tables
Match-granularity		Element-level	-	Element-level
Schema-level match	Name-Based	Name Equality; Synonyms, Hypernyms	Name Equality	Name Equality; Type Check on Input
	Structure matching	-	Similarity in structure	-
Reuse/auxiliary information used		-	Data Dictionary, Thesauri, Library of known mappings	-
Manual Work/User Input		Resolving structural conflicts; specification of some synonyms + inclusions with similarity probabilistic; validation	-	Specification of synonyms, homonyms and attribute mappings; Validation at the end
Application Area		Schema Integration	Schema Integration	Understanding of the mapping space; Evaluation of mappings; Update and maintenance of warehouses
Remarks		Algorithms to calculate new synonyms, homonyms	-	Algorithms to generate all possible mappings; Select, Project, Join, Union and Intersection

2.4 Chapter summary

This chapter gives an insight into the other related work that has been done around this problem of schema matching. It is lucid that though some amount of research has gone into this, schema matching still remains a largely elusive problem, as the applications are varies and spans several domains/ areas, and a generic solution is still evasive.

CHAPTER 3
DESIGN

3.1 Introduction

This chapter discusses the issues involved in automating the process of mapping generation, the different possible mappings that need to be taken care of, and the design of the matching algorithm.

3.2 Automation issues

As mentioned in chapter 1, automating the process of schema matching and mapping generation does not come easy, as it may seem like. Since the source schemas are defined and created independently and at various points in time, a single uniform naming convention for the attributes of the source schemas is hard to realize. Hence one needs to consider the presence of the following in the attributes of the source and warehouse schemas:

1. Synonyms
2. Homonyms
3. Attribute mappings
4. Derived attributes

These are described in detail in the following sections.

3.2.1 Synonyms

By *synonyms*, one refers to the attributes that are represented by different names in the various schemas, but which are the same attribute. For example, **SSN** of relation R1 and **SNO** of relation R2 might represent the same attribute, though referred to by different names. These attributes have different names, but the same type.

3.2.2 Homonyms

By *homonyms*, one refers to the attributes that are represented by the same names in the various relations, but which are different in structure. For example, the attributes **TYPE** of relation R1 and **TYPE** of relation R2 might be called by the same name, but which might refer to two totally disparate attributes. Here, one needs to note that the two attributes can have the same or different types.

3.2.3 Attribute mappings

This is similar to the synonyms in the respect that this represents the attributes of the warehouse relations that are referred by different names than the ones that are used to represent the same attributes in the respective source relations. For example, **EMPLOYEE_ID {varchar (10)}** of DW relation R1 and **SSN {varchar (10)}** of source relation R2 might be the same attribute, but referred to by two different names. This basically comes into the picture when the warehouse attributes need to be given a name that is more appropriate to the situation than what is there in the corresponding source schema.

3.2.4 Derived attributes

These list all the attributes of the warehouse relations that are derived or computed from more than one attribute from the source relation (s). Some examples of this type of attributes would include **Cumulative Grade Point Average**, **Grand Total** or any kind of sum, that requires more than one attribute of the source schemas to be computed from.

From this, it is clear that these issues need to be taken care of during automation, as missing out on any of these would not give a complete possible list of mappings between the warehouse and the source schemas. Hence, the system should be able to comprehend this and incorporate all the necessary changes to arrive at the correct and complete result of all the possible mappings.

3.3 Type of mappings

Various types of mappings are possible between the warehouse and source schemas. The System would have to be able to generate the complete list of all these different kinds of mappings. The following sections give some insight into the types of mappings.

3.3.1 Single source relation projection

As the name denotes, this is a selection/ projection of a single source relation. The warehouse relation can be either a complete projection of the source relation or a partial projection, as illustrated in figure 3.1.

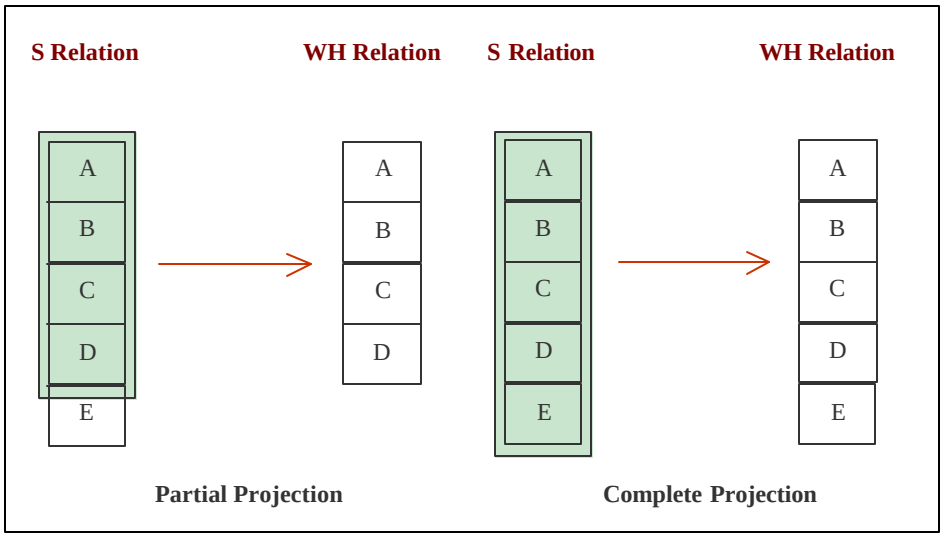

Figure 3.1. Single source relation projection.

3.3.2 Join of two or more source relations

This mapping would cover all the joins between the source relations that make up the warehouse relations. Again, here, one needs to consider two types of mappings namely, a join with join attributes and a cartesian join with no join attributes in common. One also needs to consider the kind of join – whether it is a complete or partial join of the source

relations. The joins are not limited to a simple join of pairs of relations, and would be comprehensive. This is illustrated in figure 3.2.

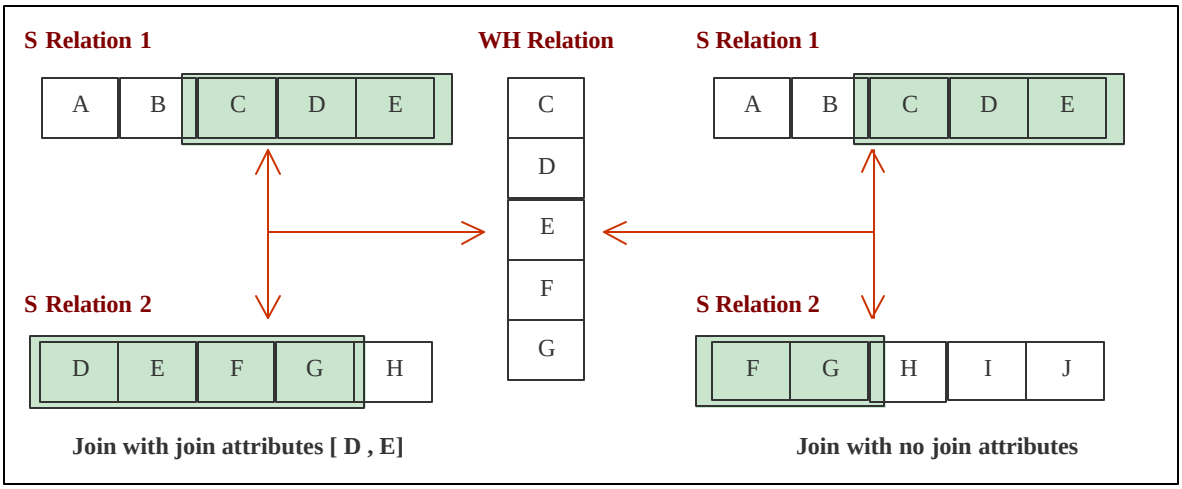

Figure 3.2. Join of source relations

3.3.3 Union/intersection of two or more source relations

This would cover all the mappings that are either unions or intersections of the source relations. One will not be able to comprehend whether it is a union or a join at this point, as one is not dealing with the tuples here. Union or intersection can be detected between source schemas if the schemas have the same set of attributes as each other and with the warehouse schema. This is illustrated in figure 3.3. Again, it can be either a complete or a partial union/ intersection of the source schemas.

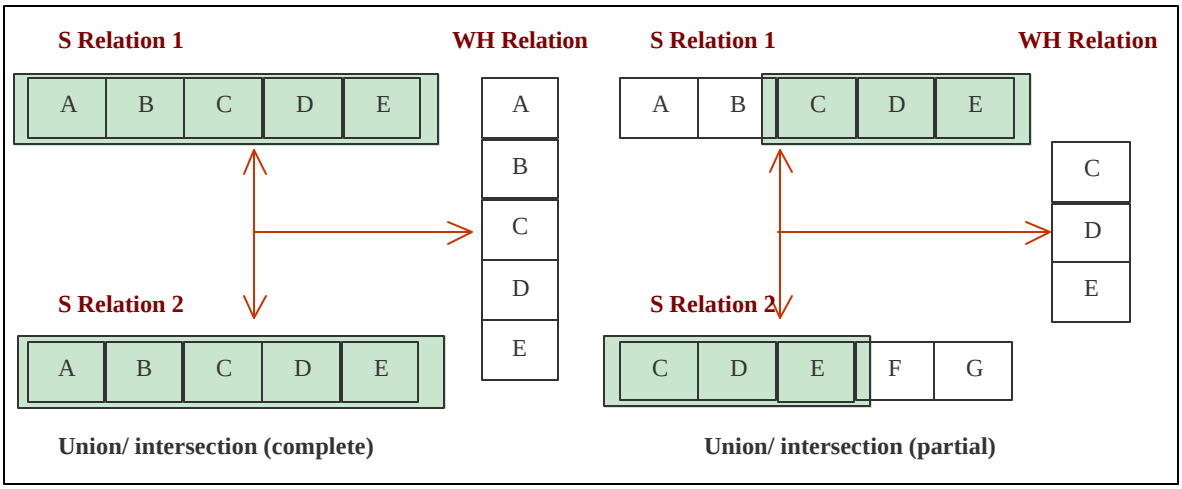

Figure 3.3. Union/ intersection of source relations

3.4 Design approach

From the various classifications of schema matching approaches [2], as shown in figure 3.4 [2], the approach that this thesis takes is the “schema-only based approach”. As described in [2], schema-level matching only considers schema information and not instance data. The information includes the usual properties of schema elements, such as name, description etc [2]. Again, under schema-based, the approach is element-level, in the sense that the match is performed for individual schema elements, which are attributes in this case. Once again, one deals with both linguistic-based and constraint-based approaches under element-level. What is carried out here is a name-based matching under linguistic matching, and a type-based matching under constraint-based matching, wherein, for each element of a relation R1, all elements of the relation R2 with same or similar name and type are identified.

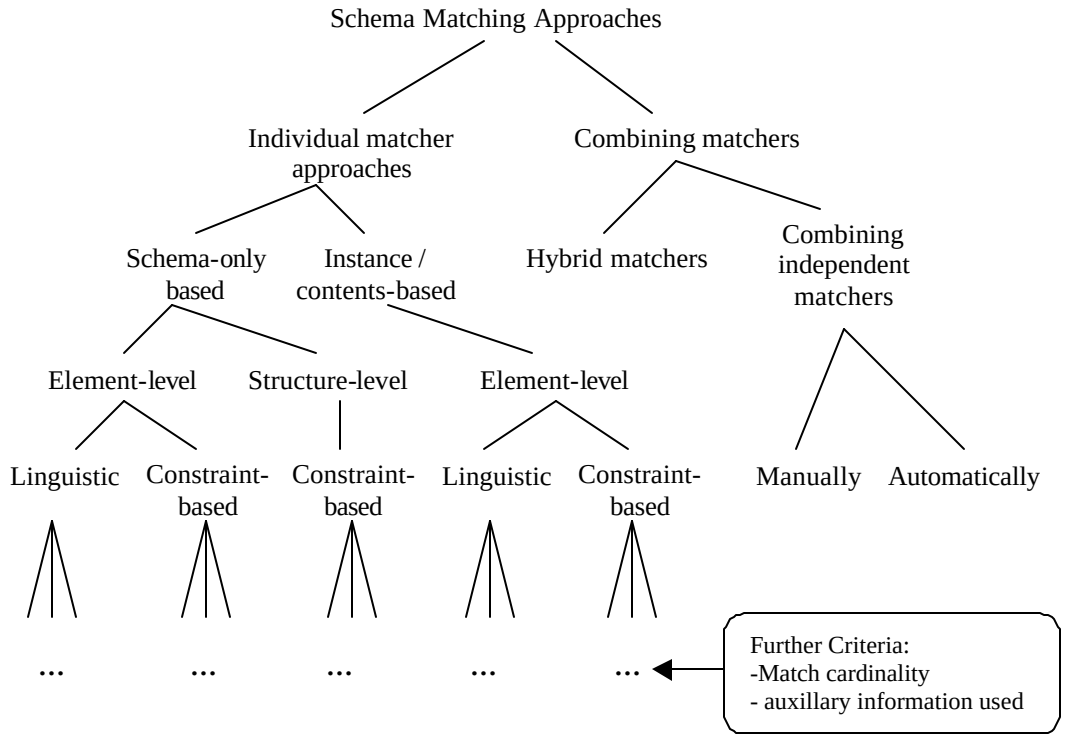

Figure 3.4. Classification of schema matching approaches.

3.5 User input

The process begins with the user providing the system with initial inputs that are crucial for arriving at the correct set of mappings between the source relations and the data warehouse relations. They are listed below in the order in which they need to be specified in the source file, the details of which are covered later in chapter 5.

The various possible inputs are:

1. The source schemas
2. The warehouse schemas
3. The list of ***homonyms***
4. The list of ***synonyms***

5. The attribute mappingsThe set of ***derived*** attributes and the corresponding source attributes

Care should be taken to stick to the same ordering of the inputs, the reason for which would be covered later in this chapter.

3.6 Design of the matching algorithm

Given a data source, the approach to creating appropriate mappings is to start by finding those elements of the source that are also present in the warehouse. This forms the match operation. After an initial mapping is created, the detailed semantics of each of the source elements need to be examined and transformations created that reconcile those semantics with those of the target. The algorithm is divided into three phases:

1. **Transformation:** This involves transforming the data for optimal generation of all possible mapping. This requires adding additional information about attributes that form part of homonyms, synonyms, and attribute mappings, as detailed out in the previous sections. The attributes in question are renamed (transformed) as required and the new names are stored separately for easy retrieval later. This would facilitate accurate and complete comparison of the attributes in the next stage namely **Intersection**, as this takes all synonyms, homonyms, attribute mappings and derived attributes into consideration. Hence starting from the next stage, the new transformed names of these attributes would be considered for comparison as against the original attribute names.
2. **Intersection**: For each of the warehouse relations, generate a list of possible source relations that have at least one attribute in common with it (the warehouse relation in question), along with the set of attributes that the relations have in common with the

warehouse relation in question. This information is stored in a separate list, the design of which will be discussed in detail in the next chapter on data structures.

3. **Mapping generation**: From the resulting subset of information obtained from the structure as described in the previous part, generate all possible mappings of the source relations to the warehouse relations, which includes select-project-join (SPJ mappings), union and intersection mappings.

3.7 Transformation

As described in the previous section, transformation takes care of the homonyms, synonyms, attribute mappings and derived attributes transforming the data for the next phase, viz. Intersection. The data from the input file is read in by the system and stored in such a way that facilitates easy storing and retrieving of the same. It is best illustrated with an example. Assume an example with two source relations R1 and R2 of sources 1 and 2 respectively. Also assume a warehouse relation R1. The set representations of the source and warehouse schemas are as shown in figure 3.5, where R1 gives the original attributes and R1' gives the transformed attributes, after applying homonyms, synonyms and the attribute mappings. One assumption here is that the warehouse relation **R1** is a partial projection of relation **R2** of **Source 2**.

SOURCE 1	RELATION R1	R1¢
	A	
	B	
	C	

SOURCE 2	RELATION R2	R2¢
	C	
	D	
	E	

DW	RELATION R1	R1¢
	C	
	D	
	E_DW	

Figure 3.5. The initial schemas.

3.7.1 Applying the homonyms

When the homonyms are read from the source file, the data is modified to accommodate these new entries. New unique names are generated for all the attributes involved. As homonyms refers to attributes that have the same name, and same or different types, but are totally different attributes, each of the attribute involved is given a newly generated name as part of transformation.

For example, assume that attribute C of relation **R1** of source 1 and attribute C of relation **R2** of source 2 are homonyms, i.e., though they have the same names, both the attributes are to be considered as separate attributes. As per the design, the names of both the attributes in this case attribute C of relation **R1** and attribute C of relation **R2** are assigned new unique generated names. This is apparent from figure 3.6, which shows the sets **R1¢** and

R2¢ for relations **R1** & **R2** being modified and the new names of "**C_001**" & "**C_002**" being assigned to the attributes C of relations R1 and R2 respectively. As shown, the mapping between the original attribute names (**R1**) and the new names (**R1¢**) is retained for future reference. Again, **R1¢** refers to the normalized set of attribute names of relation **R1**, and so forth for all the relations.

One should again note at this point that the change in the name to attribute C of relation **R2** is not reflected in the warehouse relation **R1**. This is the reason why the warehouse mapping is specified separately after specifying the homonyms and the synonyms.

SOURCE 1	**RELATION R1**	**R1¢**
	A	A
	B	B
	C	**C_001**

SOURCE 2	**RELATION R2**	**R2¢**
	C	**C_002**
	D	D
	E	E

DW	**RELATION R1**	**R1¢**
	C	C
	D	D
	E_DW	E_DW

Figure 3.6. Relations after applying the homonyms.

3.7.1.1 Issues in applying homonyms

While applying the homonyms to the initial data, two different cases need to be considered:

Case 1: Attribute has already been transformed

Assume the following example,

R1.A : R2.A

R2.A : R3.A

Where R1, R2, R3 denote the source relations and A referring to the attribute name. R2.A has already been transformed in the previous step. Hence in the next step, it is not transformed again, and only R3.A is transformed. **Case 2:** Both attributes have been transformed

In the following example,

R1.A : R2.A

R3.A : R4.A

R2.A : R3.A

R2.A and R3.A have already been transformed in the previous couple of steps. Hence, no change is necessary in the third step.

3.7.2 Applying the synonyms

When the synonyms are read from the source file, the data is modified to accommodate these new entries. For the attribute names, new names are assigned as required. The mapping between the old names and the new names are stored separately, which would facilitate easy retrieval of the original names given the new normalized names and vice versa.

For example, assume that attribute A of relation **R1** of source 1 and attribute D of relation **R2** of source 2 are synonymous, i.e., though they have different names, both the attributes are essentially the same for all matching purposes. Hence, the design involves in

retaining the name of one of the attributes, in this case the former one (of relation **R1**) as is and renaming the latter attribute (of relation **R2**), which is the same as the first attribute which would be "**A**". This is apparent from figure 3.7, which shows the normalized attribute set **R2¢** for relation **R2** being modified and the new name of "**A**" being assigned to the attribute D. Again, the mapping between the original attribute names (**R2**) and the new names (**R2¢**) needs to be retained for future cross reference.

One should note at this point that the change in the name to attribute D of relation **R2** is not reflected in the warehouse relation **R1**. This problem of transitivity is the reason why the warehouse mapping is specified separately after specifying the synonyms and the homonyms.

SOURCE 1	**RELATION R1**	**R1¢**
	A	A
	B	B
	C	**C_001**

SOURCE 2	**RELATION R2**	**R2¢**
	C	**C_002**
	D	**A**
	E	E

DW	**RELATION R1**	**R1¢**
	C	C
	D	D
	E_DW	E_DW

Figure 3.7. Relations after applying the synonyms.

3.7.2.1 Issues in applying synonyms

While applying the synonyms to the initial data, many different cases need to be considered. All the possible cases are listed as shown in table 3.1.

Table 3.1. All possible cases while applying synonyms

	ATTRIBUTE 1	ATTRIBUTE 2
1	NOT TRANSFORMED	NOT TRANSFORMED
2	NOT TRANSFORMED	TRANSFORMED – HOMONYMS
3	TRANSFORMED – HOMONYMS	NOT TRANSFORMED
4	TRANSFORMED – SYNONYMS	NOT TRANSFORMED
5	TRANSFORMED – HOMONYMS	TRANSFORMED – HOMONYMS
6	TRANSFORMED – SYNONYMS	TRANSFORMED – HOMONYMS
7	NOT TRANSFORMED	TRANSFORMED – SYNONYMS
8	TRANSFORMED – HOMONYMS	TRANSFORMED – SYNONYMS
9	TRANSFORMED – SYNONYMS	TRANSFORMED – SYNONYMS

The various possible cases have been grouped together by similarity. All the similar cases need to be handled separately, as described below:

Case 1:

Given two attributes R1.A : R2.B being synonyms,

A - Not transformed

B - 1. Not transformed (or)

2. Transformed by homonyms

This is the typical case, where, the name of attribute A is taken and the attribute B is transformed with this name of A.

Case 2:

Given two attributes R1.A : R2.B being synonyms,

A - 1. Transformed by synonyms (or)

2. Transformed by homonyms

B - 1. Not transformed (or)

2. Transformed by homonyms

In this case, the transformed name of attribute A is considered and the attribute name of B is transformed with this name.

Case 3:

Given two attributes R1.A : R2.B being synonyms,

A - 1. Not transformed (or)

2. Transformed by homonyms

B - 1. Transformed by synonyms

In this case, the transformed name of attribute B is considered and the attribute name of A is transformed with this name. This is the switch case of case 2.

Case 4:

Given two attributes R1.A : R2.B being synonyms,

A - 1. Transformed by synonyms

B - 1. Transformed by synonyms

This is an error case, which cannot be handled.

3.7.3 Applying the attribute mappings

Applying the attribute mappings to the datastructure is no different than that for synonyms and homonyms. Three different cases are possible here.

Case 1: A direct mapping from a regular attribute of a source relation to that of a warehouse relation, wherein, the attribute is referred to by a different name than what is specified in the source relation.

Case 2: A mapping between an attribute of a source relation that has been modified earlier due to the adding of *homonyms* and the corresponding attribute in the warehouse relation.

Case 3: A mapping between attributes of a source relation that has been modified earlier due to the adding of *synonyms* and the corresponding attribute in the warehouse relation.

From the earlier example, as per the assumption, attributes C, D and E of relation R2 of source 2 are projected on to the warehouse relation R1. Under attribute mappings, a one-one mapping for each of these attributes that have been changed is given. It should be noted here that there might be other attributes of the warehouse relation that are not modified by any of the cases as mentioned earlier. Such attributes are not mentioned under this section. All the three cases can be seen in the example in figure 3.7.

Case 1: Attribute E of relation R2 of source 2 is being referred by a different name in the warehouse relation R1, which is “E_DW”.

Case 2: Attribute C of relation R2 of source 2 has been modified and is now being referred to with a new name, which is “C_002”.

Case 3: Attribute D of relation R2 of source 2 has been modified and is now referred to with a new name, which is “A”.

Once the attribute mappings are read in, the changes are updated in the warehouse relation as shown in figure 3.8.

SOURCE 1	RELATION R1	R1¢
	A	A
	B	B
	C	**C_001**

SOURCE 2	RELATION R2	R2¢
	C	**C_002**
	D	**A**
	E	E

DW	RELATION R1	R1¢
	C	**C_002**
	D	**A**
	E_DW	**E**

Figure 3.8. Relations after applying the attribute mappings.

3.7.4 Applying the derived attributes

The derived attributes, which are attributes of the warehouse relation that are derived/ computed from more than one attributes of the source relations, need to be handled separately. They cannot be added to the data structure until after the generation of the mapping. Hence, the derived attributes are removed from the warehouse relations and need to be stored separately along with the attributes these are derived from.

3.7.5 Summary

Finally, at the end of transformation, the schemas would have the transformed attribute names as shown in the third column for each of the schemas in figure 3.8, which are then considered for the next phase, namely Intersection.

3.8 Intersection

For each of the warehouse relations, this stage involves in generating a list of possible source relations that have at least one attribute in common with the warehouse relation in question. This information is stored separately in a list, which at the end of this stage would contain a list of sources for each of the warehouse relations. Both name-based and type-based matches are performed between the attributes of the source and the warehouse schemas. The pseudo code for this part of the algorithm is as shown in figure 3.9.

```
for each of the warehouse relations wr_m
{
  interSet_m [] = new Set
  for each of the sources s_i
  {
    for each of the source relations r_j
    {
      aSet_j = new Set
      aSet_j = r_j intersection wr_m  (get set of intersecting attributes)

      if aSet_j != empty (implying that some common attribute exists)
      {
        add aSet_j, s_i and r_j to interSet_m[j]
      }
    }
  }
}
```

Figure 3.9. The matching algorithm - intersection.

This process is done only once initially, which facilitates the effective filtering of source relations that form no part of any particular data warehouse relation from consideration in the next step which is to generate the mapping between the data warehouse relations and the corresponding source relations, and to figure out the type of the mapping –

whether it is a projection, join, union or intersection. This process is illustrated with an example as shown in figure 3.10. Assume two source relations R1 and R2, and a warehouse relation WR1. Now, in Intersection, for each of the warehouse relations, in this case, WR1, each of the source relations are considered and the intersecting set of attributes is obtained. A non-empty intersecting set of attributes implies that the source relation in question has some attribute in common with the warehouse relation. Hence whenever a non-empty set is obtained between a pair of source and warehouse relations, the corresponding source and the set of common attributes are added to a set for each of the warehouse relations, as illustrated in figure 3.10.

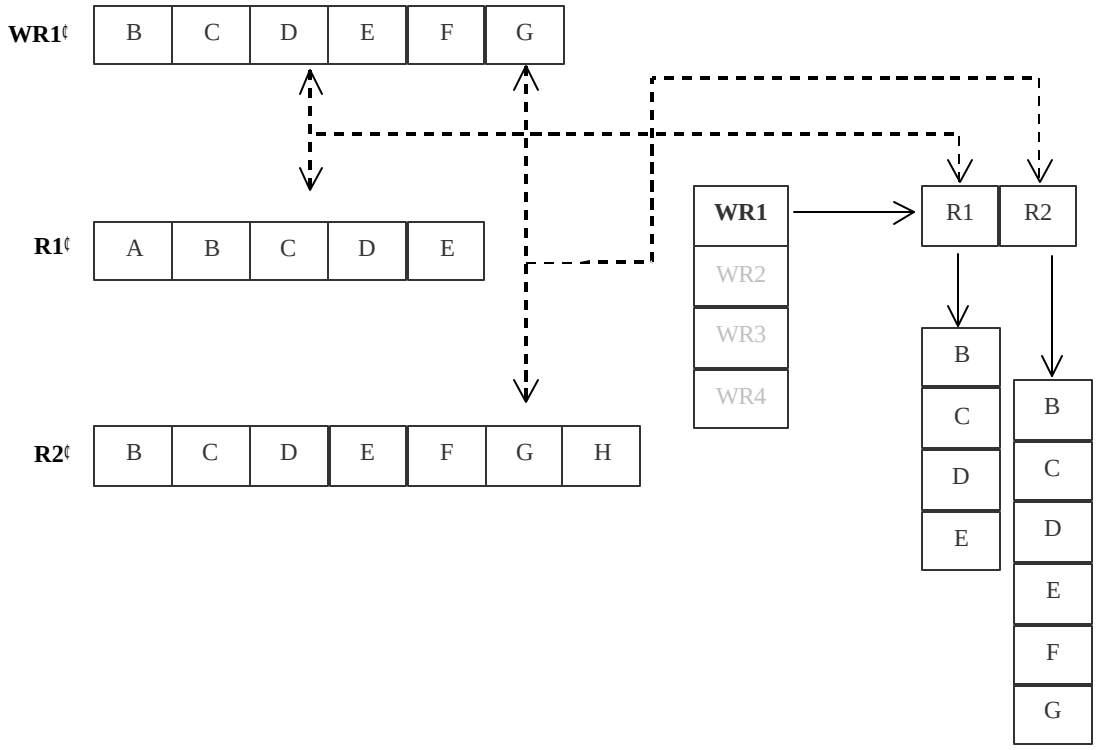

Figure 3.10. Example illustrating intersection

3.8.1 Summary

At the end of this stage, for each of the warehouse relations, one would be left with a set of source relations that have at least one attribute in common with the warehouse relation and the intersecting set of common attributes between that pair of source and warehouse relations, as described in figure 3.11.

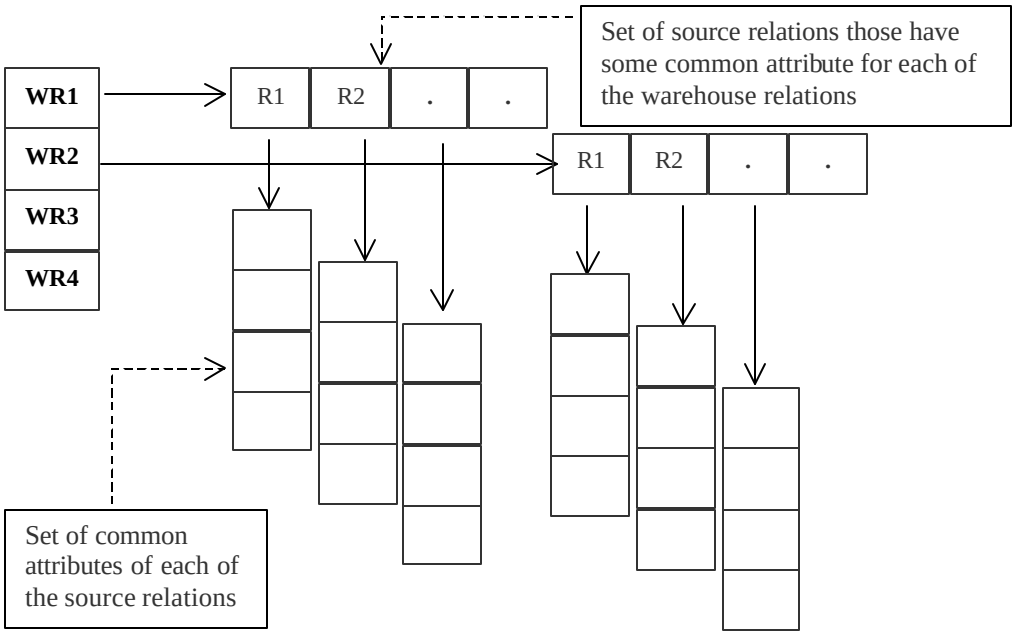

Figure 3.11. End of intersection.

3.9 Mapping generation

At the end of the previous stage, which is Intersection, for each of the warehouse relations involved, one would end up having a set of all the sources that have some attribute in common with the warehouse relations. The design in this stage involves in generating all the possible mappings between the warehouse and the source relations. One needs to comprehend at this point, that there may exist more than one source relation that has a part in deriving the data warehouse relation. So the quest here is to identify how this warehouse

relation is derived – it might be a projection of a single source relation or a join of more than one source relations, or a union or an intersection of similar source relations. For each of the warehouse relations, the sets of attributes of the source relations are retrieved and analyzed and the plausible mappings generated. The various steps/ checks involved are – one for projection and one for join and union/intersection. The set generated in the previous stage [figure 3.11] is all that is required in this stage to generate all the possible mappings. At this juncture in the design, the source relations have been filtered and only those relations that have any common attribute with the warehouse relations are considered. This considerably reduces the number of comparisons that need to be performed.

3.9.1 Projection

The pseudo code for this part of the algorithm is given in figure 3.12. As illustrated in figure 3.12, for each of the warehouse relations in question, the set of source relations that have some attribute in common with this warehouse relation is obtained from the previous stage, and for each of these relations, the set of common attributes is compared with the attribute set of the warehouse relation for equality. If they turn out to be equal, it implies that the warehouse relation is indeed a projection of that source relation.

```
 for each of the warehouse relations wr_m
{
  mapSet_m = new Set
  dwSet_m = new Set
  dwSet_m = set of attributes of the warehouse relation wr_m

// start of check for projection

  for each of the source relations  r_j in interSet_m  (refer Intersection)
  {
    get aSet_j (the set of common attributes of this relation with the warehouse
relation)

    if ( aSet_j  = = dwSet_m )
   {
     (implies that the warehouse relation is a projection of this source
relation- can be complete or partial)

     if ( (r_j intersection aSet_j) != empty
     {
       (implies it is a partial projection)
       add r_j to mapSet_m as partial projection
     }
     else
     {
        (implies it is a complete projection)
       add r_j to mapSet_m as complete projection
     }
   }
  }
// end of check for projection
```

Figure 3.12. The matching algorithm - projection.

This again, is best illustrated with an example. Consider the following example with a single warehouse relation W.R1. Assume that the source relations R1 and R2 have some attributes in common with this warehouse relation as illustrated in figure 3.13.

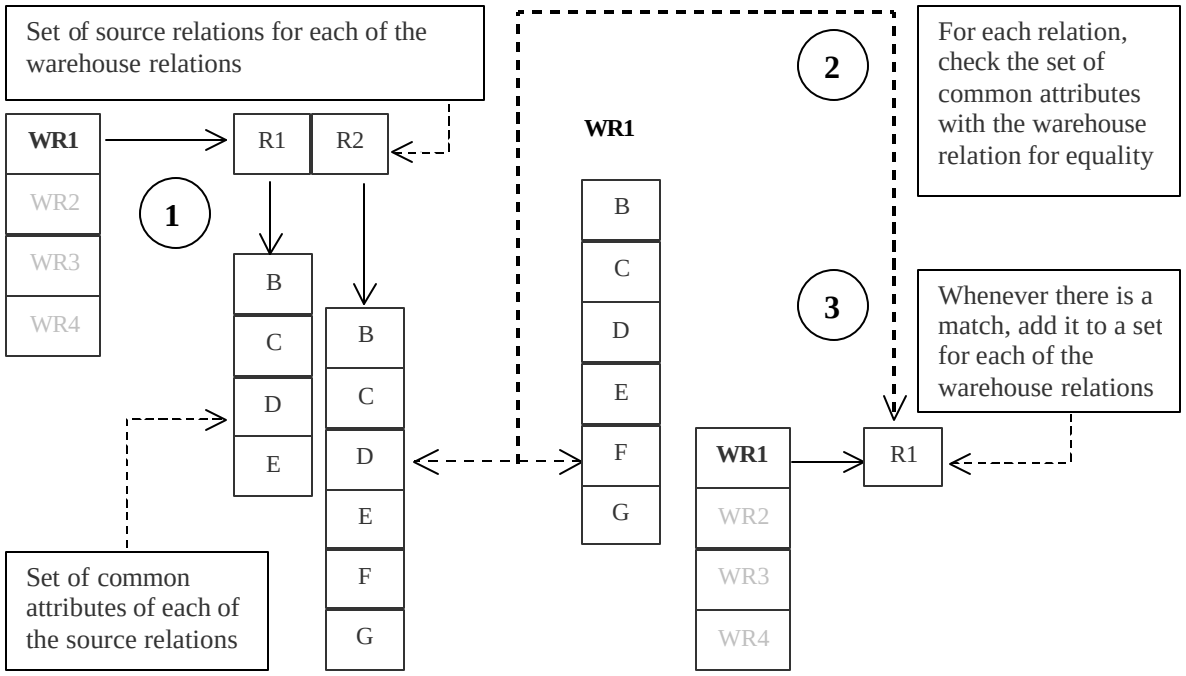

Figure 3.13. Example illustrating projection.

From figure 3.13, 1 gives the set of warehouse relations that has the set of source relations that have common attributes for each of the warehouse relations. The first step is illustrated in 2, where for each of the warehouse relations, for each of the source relation in the set, the set of common attributes is compared with the warehouse relation for equality. A match implies the presence of a projection. Whenever there is a match, as in 2, the source relation involved is added to another set for each of the warehouse relations, as illustrated in figure 3.13.

One other check that needs to be done at this juncture is the check to see if it is a complete or a partial projection. This can be done by intersecting the set of common attributes of each of the sources with the complete attribute set of the same. If the resultant set has attributes, it simply implies that it is a partial projection of the source relation, and a complete projection otherwise. This check is not stopped when a match is found, and is done

for all the source relation entries in the set for each of the warehouse relations, as one might be able to derive the warehouse relation from more than one source relations. For each of the possible projection, the source relation is added to a set as detailed out in figure 3.13.

At the end of this stage, for each of the warehouse relations, one would have a list of all the mappings that can be generated by means of projection of a single source relation, as illustrated in figure 3.14.

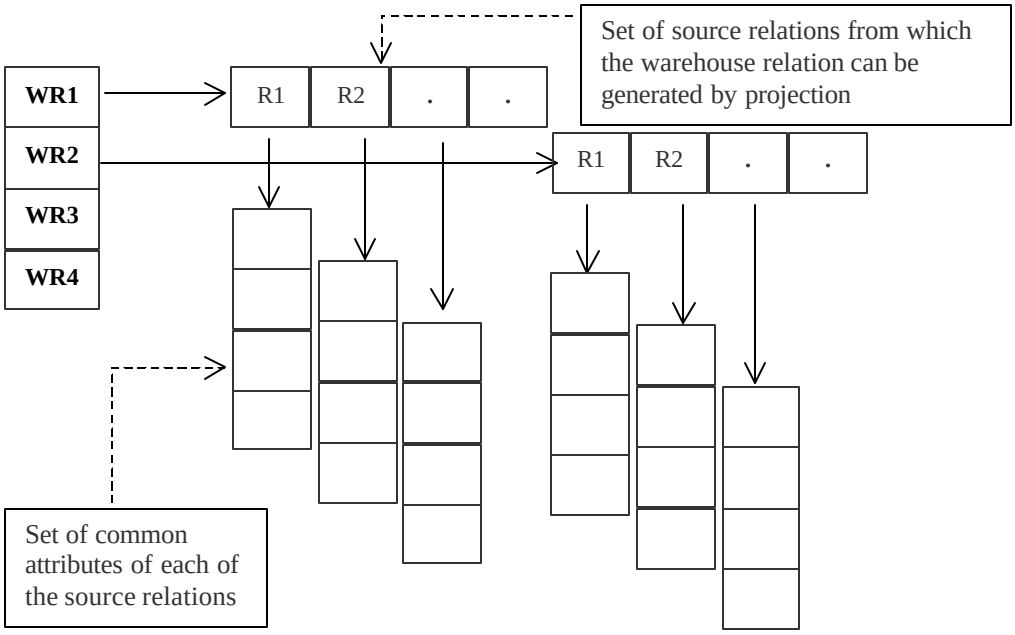

Figure 3.14. End of projection.

3.9.2 Join, union and intersection

The pseudo code for this part of the algorithm is given in figure 3.15. As illustrated in figure 3.15, for each of the warehouse relations in question, this stage involves in generating all possible pairs of source relations. For each of these pairs of relations, the combined set of common attributes is compared with the attribute set of the warehouse relation for equality. If

they turn out to be equal, it implies that the warehouse relation can be one of join, union or intersection.

```
// start of check for join, union/ projection

  for all combinations of the source relations   (r_i r_j) in interSet_m  (refer
Intersection)
  {
    get aSet_ij (the combined set of common attributes of each pair of relations)

    if ( aSet_ij  = = dwSet_m )
    {
       (implies that the warehouse relation is a join of this source relation- can
be complete or partial)

       if ( (r_i intersection aSet_i) != empty  || (r_j intersection aSet_j) != empty
       {
         (implies it is a partial projection)
         add the pair (r_i r_j ) to mapSet_m as partial projection on join
       }
       else
       {
         (implies it is a complete projection)
         add the pair (r_i r_j ) to mapSet_m as complete projection on join
       }
    }
    if ( aSet_i = = aSet_i )
    {
       (implies that it might be a join or an intersection)
       add the pair (r_i r_j ) again to mapSet_m as union / intersection
    }
  }
// end of check for join, union/ projection

}
```

Figure 3.15. The matching algorithm - join, union/ intersection

This is again best illustrated with an example as illustrated in figure 3.16.

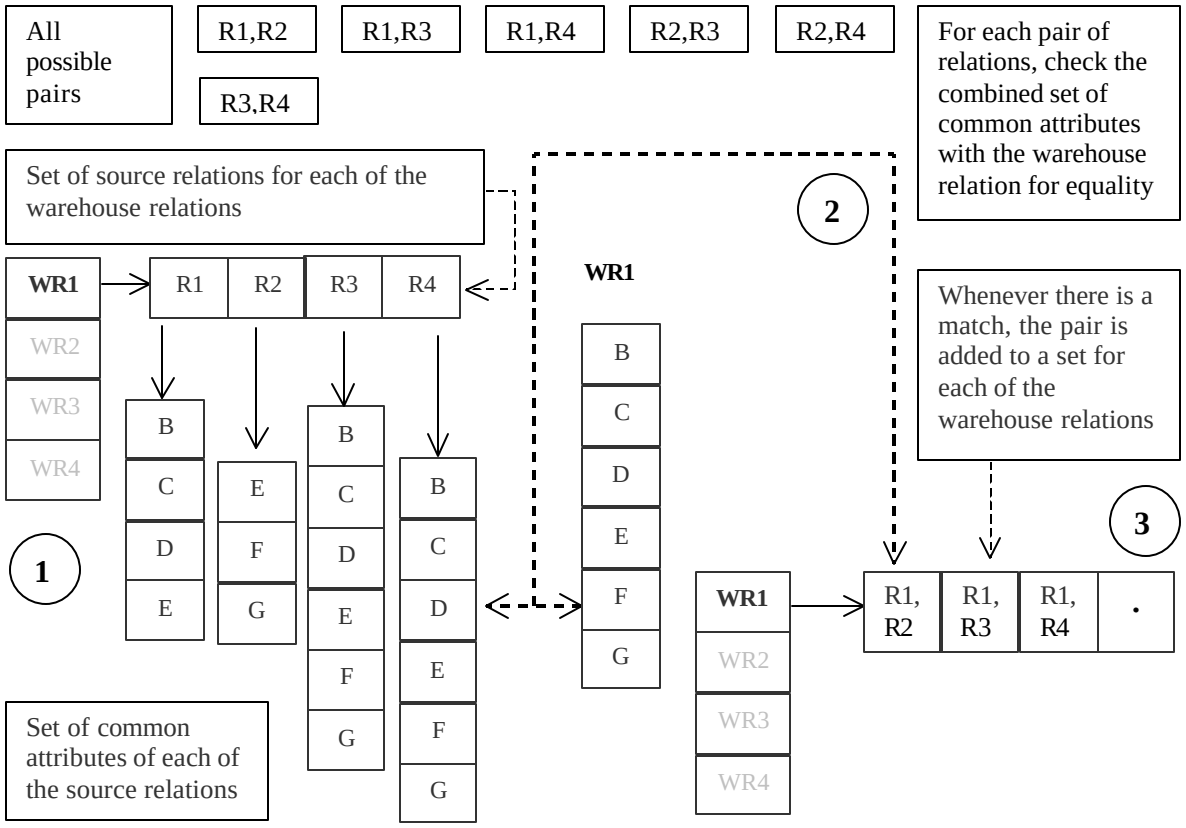

Figure 3.16. Example illustrating join.

To explain figure 3.16, this step starts with the same set of all source relations that have some attribute in common with each of the warehouse relations. The first step is to generate all possible pairs of source relations. In the example, warehouse relation WR1 has four source relations that have some attribute in common with it. Hence all the possible pairs, namely – [R1, R2] , [R1, R3] , [R1, R4] , [R2, R3] , [R2, R4] and [R3, R4] are generated. For each of these generated pairs of source relations, the combined set of common attributes is compared with the warehouse relation WR1 for equality. Equality implies that there is a possibility of a join. Again, here the check for union or intersection is an addition to this check, where in a pair can be assumed to be a union / intersection of the warehouse relation if

the set of common attributes of each of the source relations in the pair is the same, and is also the same as the warehouse relation. In the given example, the pair R3, R4 would give a possible union/ intersection, as they have the same set of attributes as shown in figure 3.16. One other check that needs to be done at this juncture is the check to see if the mapping is complete or partial.

This check is not stopped when a match is found, and is done for all the source relation pairs generated for each of the warehouse relations, as one might be able to derive the warehouse relation from more than one join of source relations. For each of the possible joins, the pair of source relations is added to a set for each of the warehouse as illustrated in 3 of figure 3.16. Another check that needs to be done here is the kind of join – a join with a common attribute (usually the key attribute), or a cartesian product, with no attributes in common. The following example illustrates multiple joins of source relations deriving the warehouse relation. Assume the set of attributes of the warehouse relation (excluding the derived attributes) in question is as follows:

Set of attributes of DW relation= { ***A B C D E F*** }

Assume that the source relations and the corresponding sets of common attributes for this DW relation are as shown in table 3.1.

Table 3.2. Example for join.

Source	Relation	Attribute Set
S1	R1	{A B C D}
S1	R2	{E F}
S2	R3	{A B C}
S2	R4	{D E F}

From the table, it is clear that both the combinations of [R1, R2] and [R3, R4] do make up the same warehouse relation shown above. This leaves us with the check for union/ intersection. Referring back to the top of this section would reveal that pairs of source relations are considered for the check for join. Now, each pair is obtained and a check of equality is done on the common attributes of the source relations of that pair. If the check turns out to be true, it implies that there is a possibility of a union or intersection.

3.9.3 Joins of more than two relations

So far, all the possible mappings that have been generated under joins, union / intersection are joins of pairs of source relations. Following the same method for generating all possible mappings of size three or more is not effective. Hence, a different approach has been followed to generate all possible mappings of size greater than two, the details of which would be covered in the following sections.

To start with, one has the set of source relations that have some attribute in common with each of the warehouse relations as shown in figure 3.14. One also requires the maximum length of joins that need to be generated for the give warehouse relations. Assume N as the number of source relations for the warehouse relation WR1, and M being the maximum length of joins that need to be generated. Since one starts with joins of pairs of relations, the remaining joins that need to be generated are from three onto M. The process involves in generating all possible combinations of source relations of length three up to M. For each combination generated, check is done to see if there is a possibility of a join, union/ intersection there. But for sake of explanation, the two stages of generating the possible combinations and checking for join have been split into two separate sections.

3.9.3.1 Generating all possible combinations

This follows a compare and elimination algorithm, which reduces the number of comparisons to a minimum as against an exhaustive comparison. In the first pass, the check is between a set of the source relations in question for each of the warehouse relations, and the complete set of all the possible pairs of relations that have already been generated. Now, all the possible joins of length three are generated. The subsequent pass would do a comparison between the same set of source relations and the set of newly generated joins of length three and so on, until the limiting length of M is reached. This is best illustrated with an example.

Assume a warehouse relation WR1 with four source relations (N) having some attribute in common with it. Also assume that one is required to generate all the possible joins of all possible lengths. So, M in this case would be four, which would be the maximum length of the join that is possible. Now, figure 3.17 details out the comparison and elimination process for the first pass, and the generation of all the joins of length three. From figure 3.17, it is clear that the process of comparison and elimination has reduced the number of comparisons considerably.

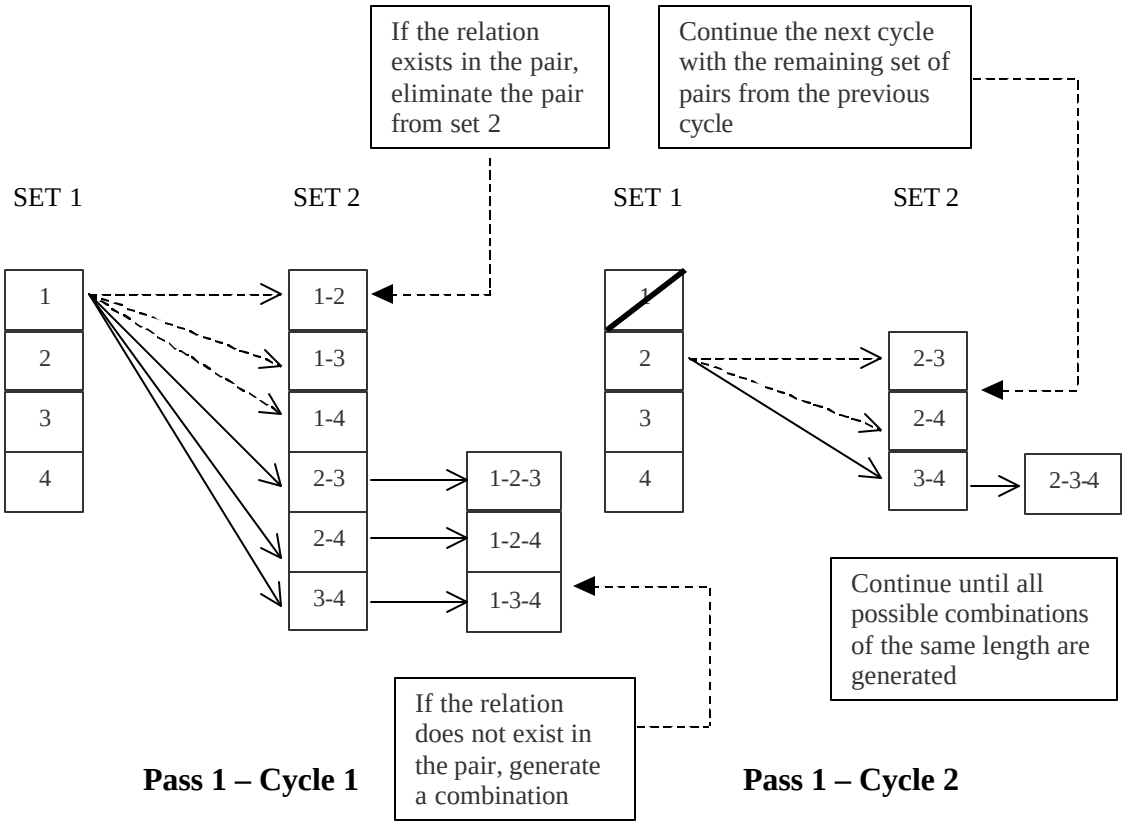

Figure 3.17. Illustrating generation of multiple joins – pass 1

Now, in the next pass, as mentioned earlier, the set of relations is compared with the set of generated joins of length three to generate all joins of the next length, viz., four. The second pass is illustrated in figure 3.18.

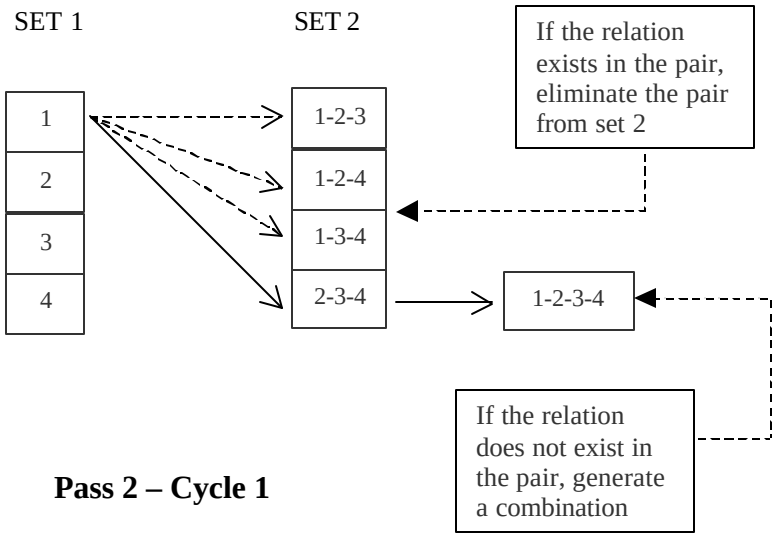

Figure 3.18. Illustrating generation of multiple joins – pass 2

Figure 3.18 illustrates the second pass, where the comparison is between the same set of source relations and the set of newly generated joins of length three, to generate joins of length four in a similar way as described for pass 1. This is repeated until all possible joins have been generated.

3.9.3.2 Check for possible mappings

As mentioned at the start of this section, the check for a possible mapping is done right after the generation of the joins. The process of checking for mapping is described in this section.

For each generated join, assume **n** as the length of the join. Now, taking all the relations that make up the join into consideration, the list of all the pairs of these relations are obtained. For each of these pairs of relations, a check is done to see if the intersecting set between them is not empty. If it is not, then a counter is incremented. At the end of the cycle, the counter is compared with a pre-set threshold [**(n-1)(n-2)/2 + 1**]. There would be a

possible mapping if the counter value is greater than or equal to the threshold value. The threshold value makes sure that all the relations involved in the join have been considered, and none have been left out. Figure 3.19 gives an example to illustrate this.

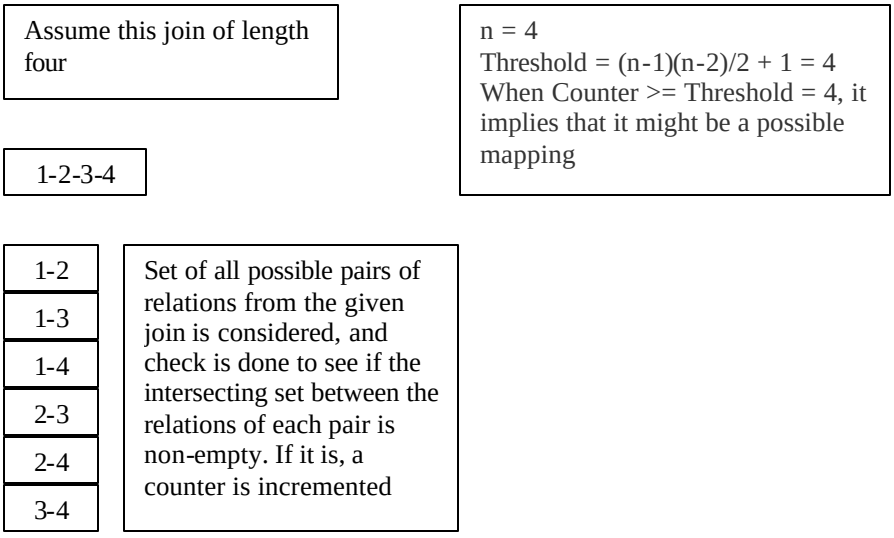

Figure 3.19. Example Illustrating check for mapping

Again, when a possible mapping is found, as before, it is added to the set of mappings for each of the warehouse relations. As in the check for pairs, all the checks are done here to check for join, union/ intersection, complete/ partial join and joins with/ without join attributes as described in the previous section 3.9.2.

3.9.4 Summary

At the end of this stage, for each of the warehouse relations, one would have the complete set of all the possible mappings including projection, pair-wise and joins of lengths more than two, unions/ intersections, as illustrated in figure 3.20.

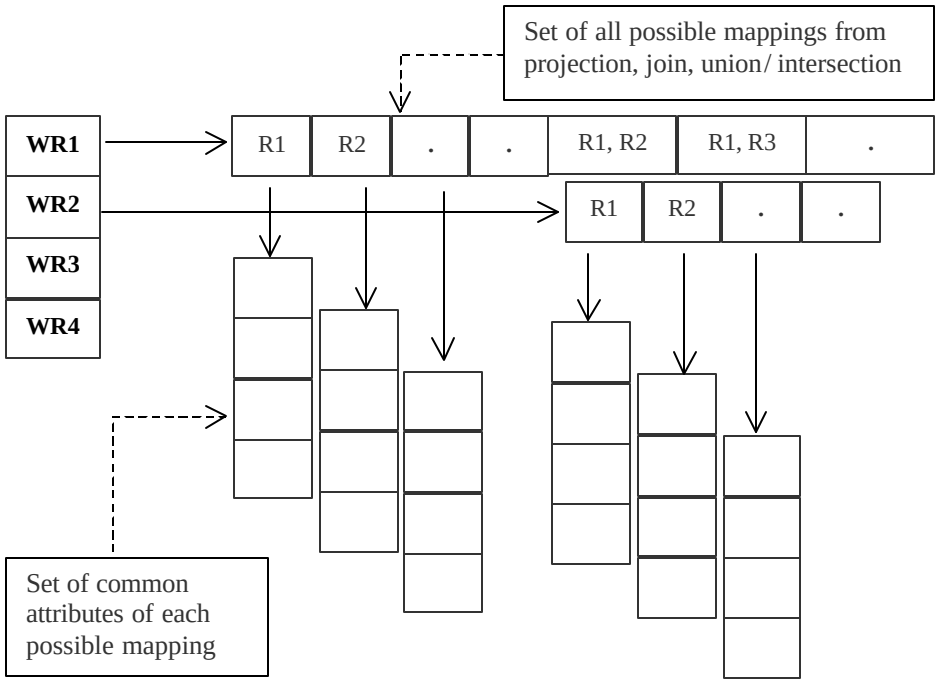

Figure 3.20. End of mapping generation phase.

3.10 User validation

Now, it is apparent that more than one possible mapping would be possible for each of the warehouse relations. Hence, to overcome this ambiguity, all the possible mappings for each of the warehouse relations are presented to the user, and the user finally validates the results. It is only after the user validation, that execution proceeds on to generating the appropriate code for setting the triggers for updates and maintenance of the warehouse relations.

3.11 Reverse transformation

Once the mappings have been generated, the process of reverse-transformation returns the attribute names that have been transformed back to the original form to facilitate

the generation of code for setting the triggers on the source schemas to propagate updates to the warehouse schemas.

3.12 Chapter summary

In this chapter, one looked at the issues involved in automating schema matching and mapping generation. All the various possible mappings that were considered were also discussed. Then, the discussion went on to talk about the matching algorithm, detailing out the various stages in the algorithm and the issues involved. The next chapter would look at the data structures used for storing and retrieving the data and for generating the mappings.

CHAPTER 4
DATASTRUCTURES IN DETAIL

4.1 Overview

The various data structures required for effective storing and retrieving data are explained in detail in this chapter.

4.2 Initial data structure

The *Initial data structure* forms the base for the whole design to rest on. It stores all the information furnished by the user as discussed in the previous chapter. It typically stores:

1. Various *Schemas* (source and data warehouse) and their specifications
2. The *homonyms*
3. The *synonyms*
4. The attribute mappings
5. The *derived attributes*

4.2.1 Description

The data structure, [figure 4.1] consists of a hashtable named ***"dwdatabases"*** which would serve as the outer most structure that would have the list of sources and data warehouses, with the name of the databases as the key and a ***vector*** as the value. This vector in turn consists of these three elements:

Vector[0] – A ***Boolean*** that is set to true or false depending on whether the database in question is trigger-based or difference-based respectively

Vector[1] – A ***string*** value that has the DBMS information for the given source

Vector[2] – A ***Hashtable***

This hashtable in-turn has the relation names as the key and a ***vector*** as the value for each of the sources. This vector will have two elements:

Vector[0] – This points to a ***hashtable*** again, that contains the attribute names in the key field and a ***vector*** in the value for each attribute of the relation in question, and this vector in turn contains from two elements to many depending on the type of attribute. The various cases possible are listed below:

Case 1. A normal attribute of a relation:

Vector[0] – The type of the attribute (as a String)

Vector[1] – Field to specify whether it is a key or a derived attribute (would be null in this case)

Vector[2] – An optional field that would be added if the current attribute needs to be renamed later *(discussed later)*

Case 2. A key attribute of a relation:

Vector[0] – The type of the attribute (as a String)

Vector[1] – Field to specify whether it is a key or a derived attribute (would have "KEY" in this field)

Vector[2] – An optional field that would be added if the current attribute needs to be renamed later *(discussed later)*

Case 3. A derived / computed attribute of a warehouse relation:

Vector[0] – The type of the attribute (as a String)

Vector[1] – Field to specify whether it is a key or a derived attribute (would have "DERIVED" in this field)

Vector[1] – This points to another ***hashtable*** that contains the new names of the attributes in the key fields and the old names of the same attributes in the value fields. *(This would be discussed in detail later)*

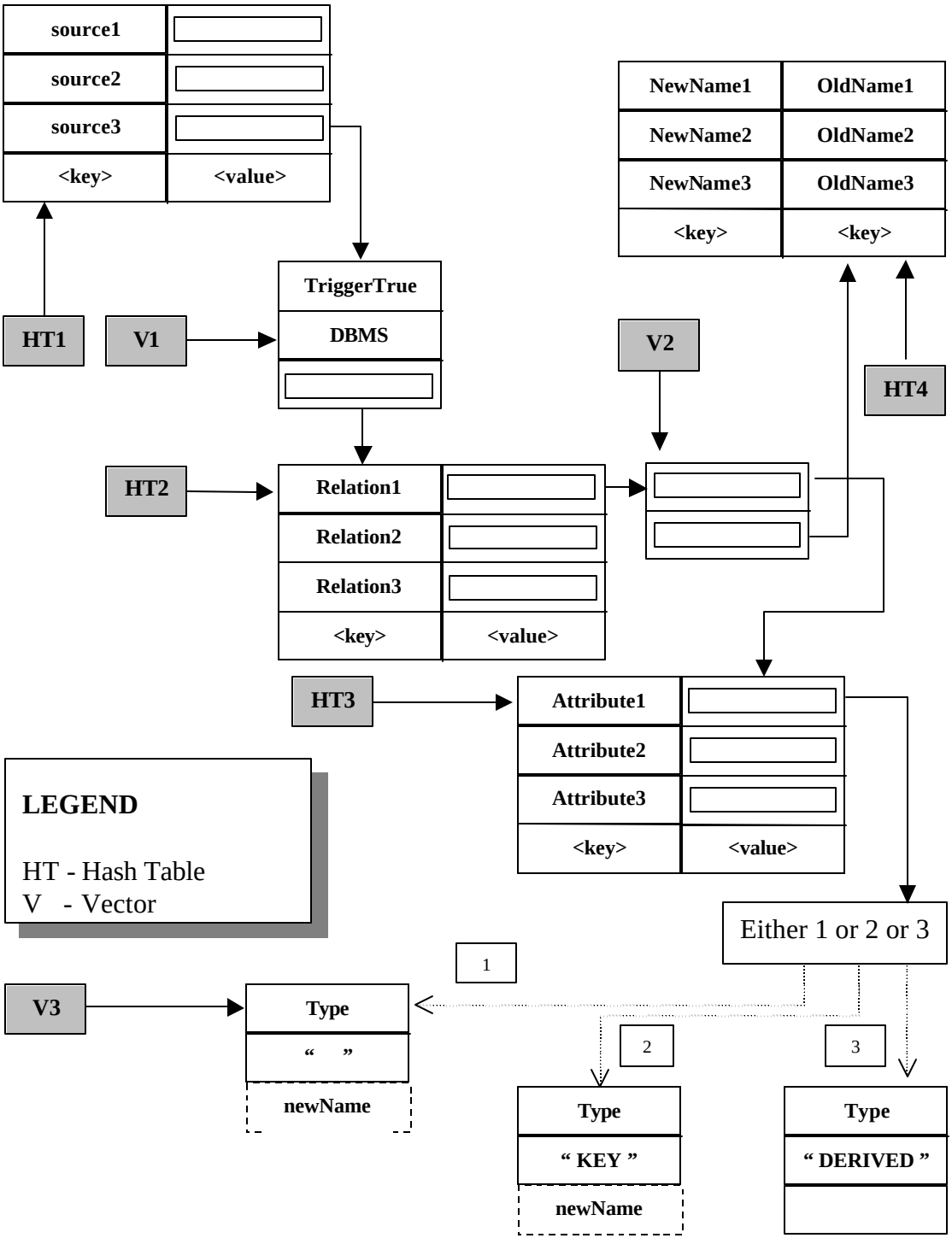

Figure 4.1. Initial data structure.

4.3 Data structure intersection

In addition to the main data structure, another data structure [figure 4.2] was needed to save the set of intersecting attributes between any (source relation – data warehouse relation) pair, which would be required later on to set the triggers appropriately. This structure can be described as:

4.3.1 Description

A ***vector*** named "***intersectVector***" the will contain one element for each of the relations of the data warehouse (assuming a single data warehouse for now). Each element of this vector is another ***vector*** that in turn contains four to five elements as shown:

1. The ***name*** of the data warehouse
2. The ***name*** of the data warehouse relation
3. An ***integer*** value that gives the count of the number of source relations that have attributes in common with this warehouse relation.
4. A data Structure "***AttribSet***" (described next)
5. Another ***AttribSet*** if required (for union or multiple partial projection)

In the case where multiple source relations are involved, additional elements are added to the vector as shown above. The data structure ***AttribSet*** mentioned above has the following elements:

1. **Name** of the source database
2. **Name** of the source relation
3. **Set** of intersecting attributes (between this source relation and the data warehouse relation in question)

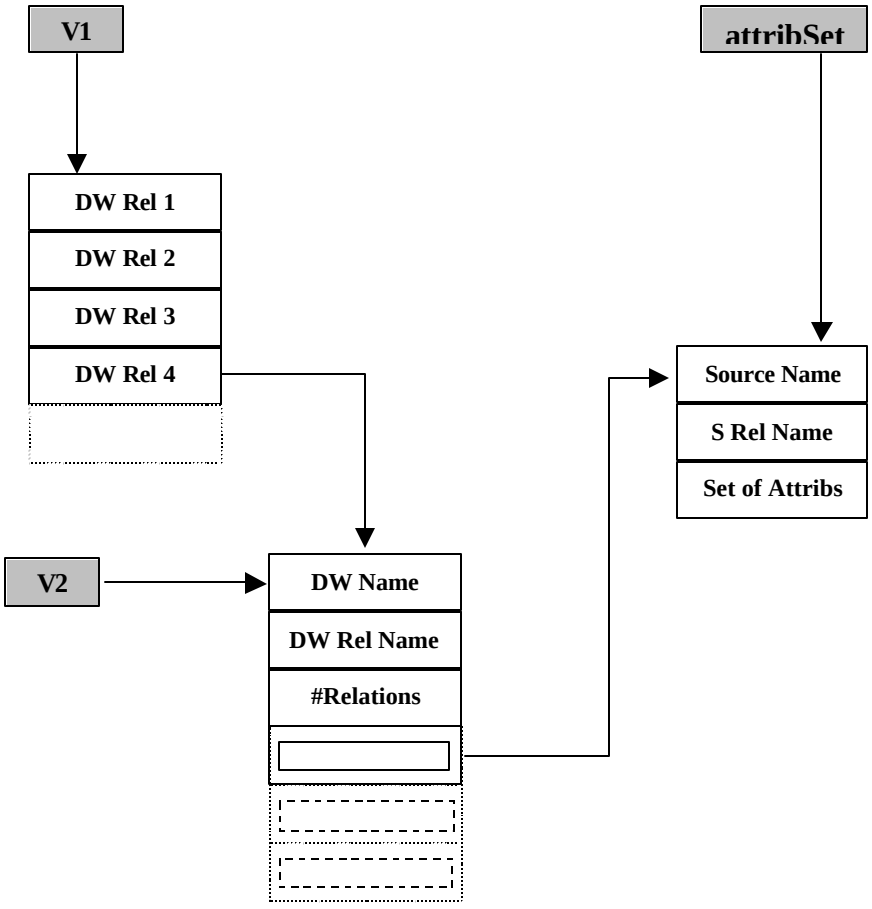

Figure 4.2. Data structure that stores the set of intersecting attributes.

4.4 Data structure check vector

The data structure "***checkVector***" is the structure that will host the final required mappings between the source and the data warehouse relations. There is one instance of checkVector for each of the data warehouse relation in question. This structure consists of a ***vector v1*** that has the total number of elements equal to the number of possible mappings between the source relations and the data warehouse relation. The type of mapping, whether it is a single source relation projection, or a join of two source relations, or a union or

intersection is defined by another ***vector v2*** [figure 4.3]. This vector V2 has the following elements:

1 – A ***string*** field that would have either “complete” or “partial” as its value, depending on whether the mapping is a complete or partial projection of the source relation, join or union, whatever the case maybe.

2 – The source relation information, i.e., the ***attribSet*** structure of the source relation (refer previous section) for a single source relation in case of projection, and more than one as required for a join, union or intersection.

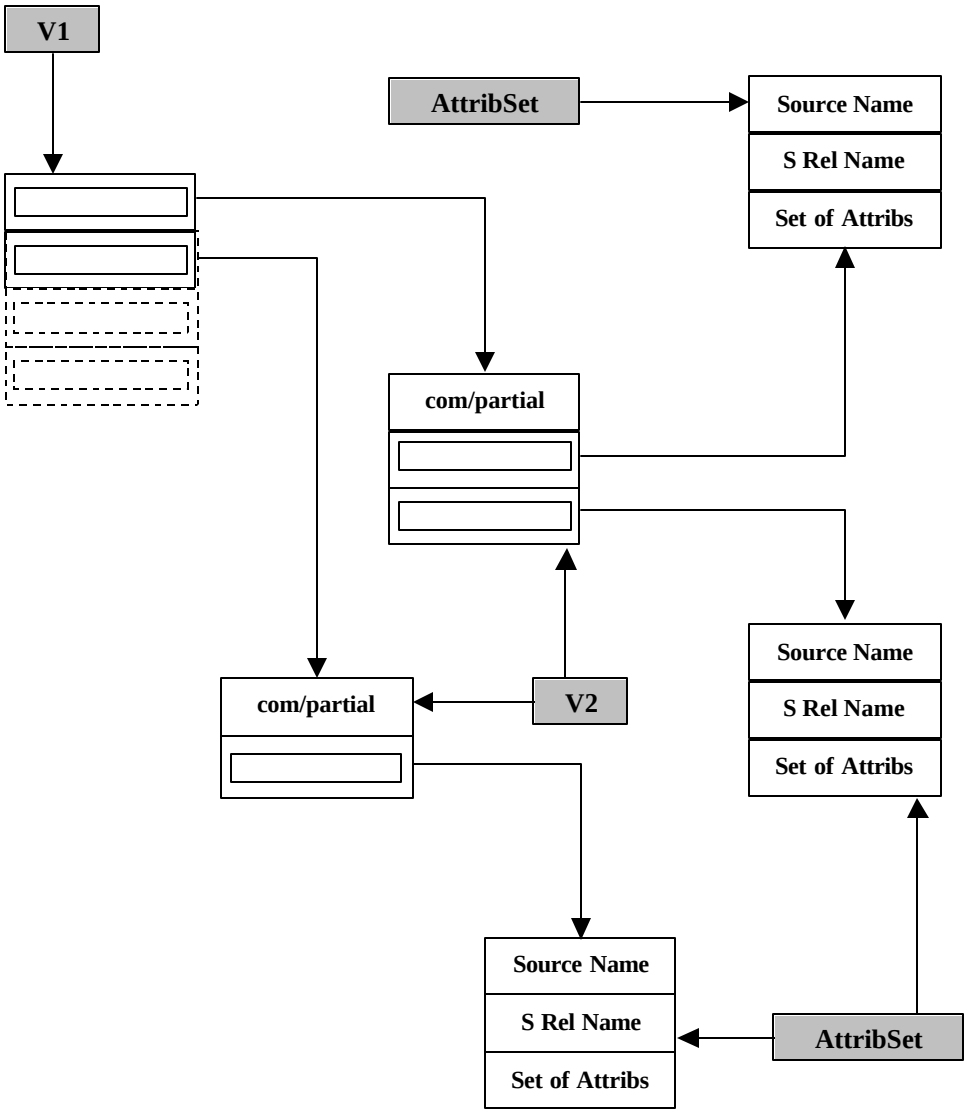

Figure 4.3. Structure that holds the final generated mappings.

4.5 Chapter summary

In this chapter, one looked at the various structures that made the realization of the design possible in this thesis. The next chapter goes into the design implementation in detail, and talks about the algorithm and the issues involved in implementation.

CHAPTER 5
DESIGN IMPLEMENTATION

5.1 Overview

This chapter discusses about the design implementation in detail, starting with the description of the source file provided by the user to how the data is read and stored, and the implementation of the schema-matching algorithm in detail. This also deals with the different implementation issues involved in the same and the methods used to overcome them.

5.2 User input

The input is read from a source file given by the user as explained in chapter 4. Figure 5.1 provides us with an accurate representation of the source file.

schemas::

source <S name> <true/false> <platform>
relation <R name>
attribute <A name> <type> {key/Derived}
attribute <A name> <type> {key/Derived}
.
.
relation <R name>
attribute <A name> <type> {key/Derived}
attribute <A name> <type> {key/Derived}
.
.
source <S name> <true/false> <platform>
relation <R name>
attribute <A name> <type> {key/Derived}
attribute <A name> <type> {key/Derived}
.
.
warehouse <DW name>
relation <R name>
attribute <A name> <type> {key/Derived}
attribute <A name> <type> {key/Derived}
.
.
homonyms::
<A name> :: <S name>.<R name> ; <S name>.<R name>
.
.
synonyms::
<S name>.<R name>.<A name> :: <S name>.<R name>.<A name> ; <S name>.<R name>.<A name>
<S name>.<R name>.<A name> :: <S name>.<R name>.<A name>
.
.
attributemapping::
<DW name>.<R name>.<A name> :: <S name>.<R name>.<A name>
.
.
derived::
<DW name>.<R name>.<A name> :: <S name>.<R name>.<A name> ; <S name>.<R name>.<A name>
.

Legend:

S name – Source Name
R name – Relation Name
A name – Attribute Name
DW name – Warehouse Name

Figure 5.1. Source file.

The keywords "**schemas**", "**synonyms**", "**homonyms**", "**attributemapping**" and "**derived**" make the design perform various operations when encountered. For example, when the keyword "**schemas**" is encountered, the system knows that it will be followed by a set of schemas, source or warehouse. Likewise, the sub keyword "**source**", "**relation**" and "**attribute**" make it easier for the system to identify the type of input that it is reading in. A more precise description of the various parts of the source file provided by the user is given below. This is subdivided into 5 sections, analogous to the source file [figure 5.1] as split up by the keywords.

5.2.1 Schemas

This refers to the source and the data warehouse schemas that the user wants to use. The syntax for specifying the schema in the source file [figure 5.1] is as shown below:

For the source,

source <S name> <true/false> <platform>

where,

<*source name*> gives the name of the source relation

<*true/false*> specifies whether the source is trigger-based (true) or difference based (false)

<*platform*> gives the DBMS platform e.g. Oracle on NT

For the relation,

relation <R name>

where,

<*relation name*> gives the name of the relation

and for the attribute,

attribute <A name> <type> {key/Derived}

where,

<attribute name> gives the name of the attribute

<type> gives its type (char, varchar, integer etc)

{Key/Derived} is an optional field that is specified as "*KEY*" or "*DERIVED*" if that attribute is a key or a derived attribute (for data warehouse relations only) respectively

The data warehouse schemas are specified in a similar manner, except that the name "source" is replaced with the name "warehouse" [figure 5.1].

5.2.2 Homonyms

Under homonyms, we list all the attributes of the source relations that are referred to by the same name, but that are dissimilar. The syntax for specifying the homonyms in the source file [figure 5.1] is as shown below:

<A name> :: <S name>.<R name> ; <S name>.<R name>

To state an example,

s1.r1.attribute1 :: s2.r1.attribute4 ; s3.r2.attribute2

which means, *attribute1* of relation *r1* of source *s1*, is the same as (*or homonymous to*) *attribute4* of relation *r1* of source *s2* and *attribute2* of relation *r2* of source *s3*.

5.2.3 Synonyms

Under synonyms, we list all the attributes of the source relations that have different names, but are the same. The syntax for specifying the synonyms in the source file [figure 5.1] is as shown below:

<S name>.<R name>.<A name> :: <S name>.<R name>.<A name> ; <S name>.<R name>.<A name>

To state an example:

s1.r1.attribute1 :: s2.r1.attribute4 ; s3.r2.attribute2

which means, *attribute1* of relation *r1* of source *s1*, is the same as (*or synonymous to*) *attribute4* of relation *r1* of source *s2* and *attribute2* of relation *r2* of source *s3*.

5.2.4 Attribute mapping

Under attribute mapping, all the attributes of the data warehouse relations that are referred to by different names as against their analogous counterparts in the source relations are listed, and the mapping between them is given. The syntax for specifying the attribute mapping in the source file [figure 5.1] is as shown below:

<DW name>.<R name>.<A name> :: <S name>.<R name>.<A name>

To state an example,

dw1.r1.attribute1 :: s2.r1.attribute4

which means, *attribute1* of relation *r1* of datawarehouse *dw1*, is the same as *attribute4* of relation *r1* of source *s2*. As explained in chapter 3, three unique kinds of mappings are possible here:

Case 1: a direct mapping from a regular attribute of a source relation to that of a warehouse relation, wherein, the attribute is referred to by a different name than what is specified in the source relation.

Case 2: a mapping between an attribute of a source relation that has been modified earlier due to the adding of *homonyms* and the corresponding attribute in the warehouse relation.

Case 3: a mapping between attributes of a source relation that has been modified earlier due to the adding of *synonyms* and the corresponding attribute in the warehouse relation.

5.2.5 Derived

Under derived, all the attributes of the data warehouse relations that are derived or computed from more than one attribute of the source relations are listed. The syntax for specifying the derived attributes in the source file [figure 5.1] is:

<DW name>.<R name>.<A name> :: <S name>.<R name>.<A name> ; <S name>.<R name>.<A name>

To state an example,

dw1.r1.attribute1 :: s2.r1.attribute4 ; s2.r1.attribute5 ; s2.r1.attribute6

which means, *attribute1* of relation *r1* of datawarehouse *dw1*, is derived from attributes *attribute4, attribute5 & attribute6* of relation *r1* of source *s2*

5.3 Storing the data

The data is read in from the *source file* [Figures 5-1]. Once the source file is open, depending on the keywords read in, various operations are performed accordingly. The data is stored in a datastructure named "*Initial datastructure*", that is described in detail in chapter 4 [figure 4-1]. This structure stores the data in a hierarchical manner, with the sources at the top level, followed by the relations of the sources, followed by the attributed of the relations. This structure holds good for the warehouse relations too. The process of storing the various parts of the source file is explained in detail in the following sections, along with the issues involved in the implementation of the same, and the solution arrived at.

5.3.1 Storing the source and warehouse schemas

This is a straightforward implementation of the design, wherein, the source characteristics are read in from the source file and the datastructure updated accordingly. Since the file stores the schema information in a hierarchical order, the storing of the same is done in a similar manner, starting with the sources, and going down to the relations and the attributes of each of the relations and so on.

```
source S1 true ORACLEOMEGA

relation STUDENTINFO

attribute SSN char(9) key
attribute FIRSTNAME varchar(15)
attribute LASTNAME varchar(15)
attribute AGE int
attribute GENDER char(1)
attribute NATIONALITY varchar(15)
attribute RACE varchar(15)
attribute ADDRESS varchar(30)
attribute HOMATT1 varchar(10)
attribute HOMATT3 int
attribute SYNATT1 varchar(30)
attribute SYNATT4 char(20)
```

Figure 5.2. Schema part of a sample source file.

The storing of the data is best illustrated with a live example. Consider a source file, a part of which is shown in figure 5.2. Figure 5.3 shows the initial datastructure at initialization, before any data is added to it.

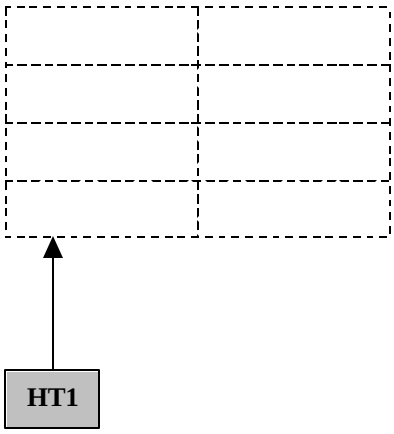

Figure 5.3. Storing source/ warehouse schemas - stage 1.

Figure 5.4 shows the same structure after the source characteristics are read in from the source/input file, which is the first line of figure 5.2.

source S1 true ORACLEOMEGA

A new element has been added to the hashtable HT1 with the source name "S1" as the key and a new Vector "V1" as the value. Again, two elements are added to this vector, and the values "true" and "oracleomega" are added to the first and the second elements of the vector V1, representing whether the source is trigger/difference based (true being trigger-based) and the platform and the RDBMS where the source resides in that order. A new empty hashtable is added as the third element of the vector V1 for the relations of this source.

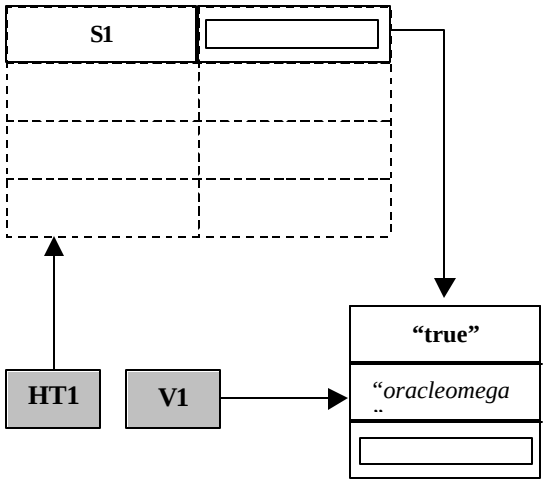

Figure 5.4. Storing source characteristics.

When the next line is read in, which has the keyword "relation", a new element is added to the newly added hashtable "HT2" with the relation name as the key as shown in figure 5.5. For the value, a new vector V2 is added. Now this vector will have a single element to start with, but another one might be added later on. This would be covered in detail in the following sections, starting with section 5.3.2 covering the storing of synonyms.

For now, a new hashtable HT3 is added as the first element of this vector V2, as shown again in figure 5.5.

Now, when the attributes are read in line by line, for each of the attributes, a new entry is added into the hashtable HT3 as shown in figure 5.6, with the attribute name as the key and a new vector V3 as the value. Again, V3 has only two elements to start with, and the third one is added as when necessary. This is also covered in detail in the subsequent sections. For now, based on the type of attribute- either a normal, key or a derived attribute (for attributes of the warehouse relations only), the vector V3 will have either " " or "key" or "derived" as the second element, with the attribute name as the first. This too is shown in figure 5.6, for the given example. Two attributes are shown added to the hashtable.

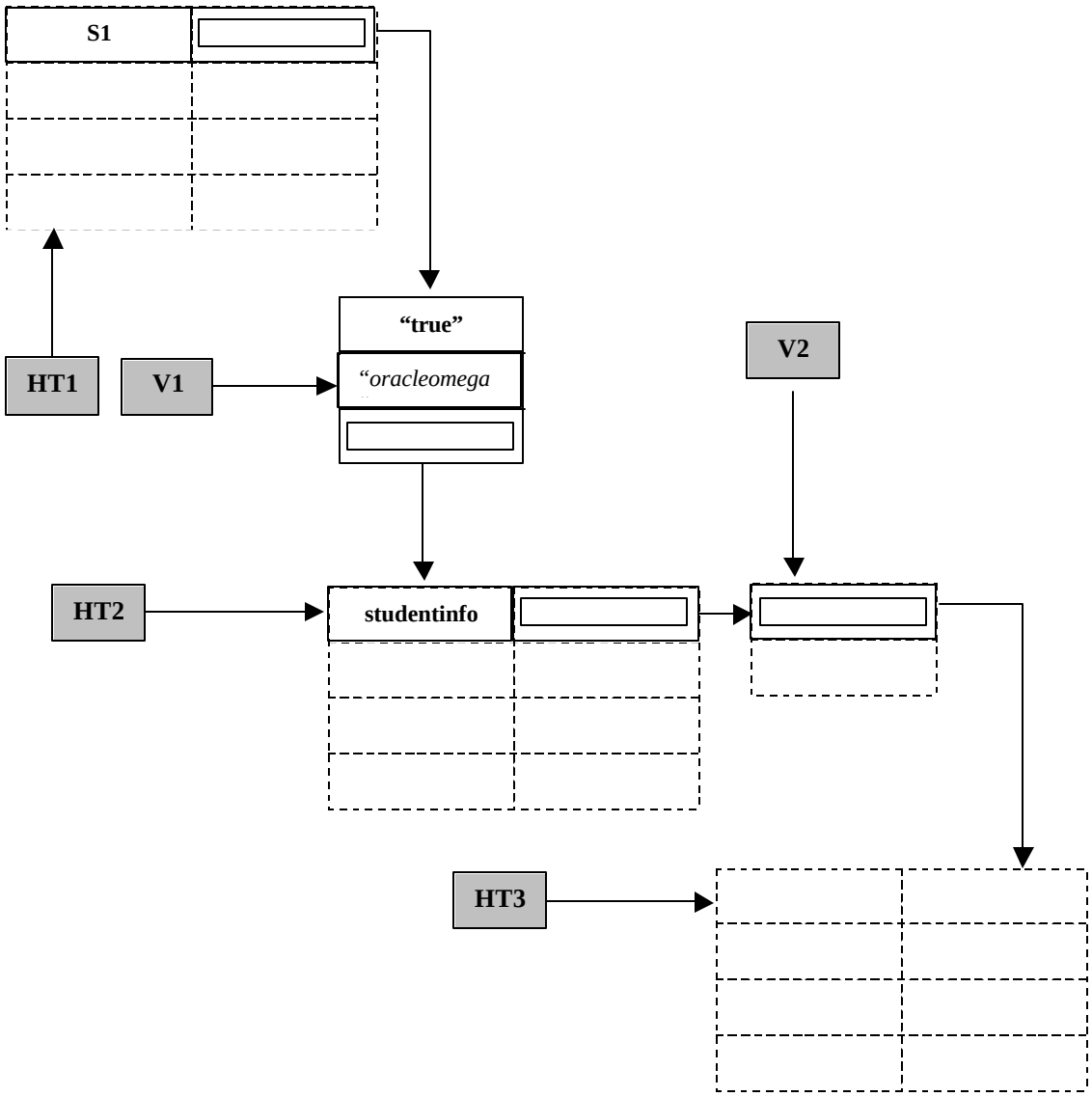

Figure 5.5. Storing relation characteristics.

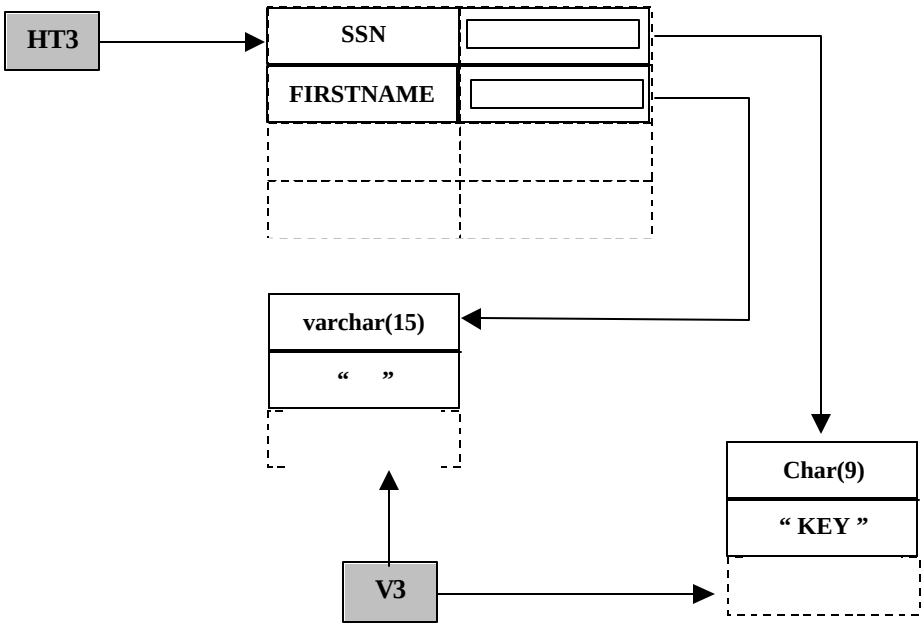

Figure 5.6. Storing attribute characteristics.

Once the source schemas are added, the same process is repeated for the warehouse schemas. One needs to note here that the source and the warehouse schemas are stored in the same structure to avoid unnecessary wastage of space. Once the source and the warehouse schemas are stored in the datastructure as described in this section, the synonyms are read in and are stored as described in the next section.

5.4 Transformation

This forms the first stage of the matching algorithm where in the data that has been read in from the source file is transformed to include the homonyms, synonyms, attribute mappings and the derived attributes. The following sections detail out the implementation and the issues involved in applying all these to the initial data.

5.4.1 Applying the homonyms

When the homonyms are read from the source file, demarked by the keyword "***homonyms***", the data structure is modified to accommodate these new entries. Again, for easy comprehension, assume a source file, part of which is shown in figure 5.7.

```
homonyms::
HOMATT1 :: S1.STUDENTINFO   ; S2.COURSETAKEN ; S3.STAFFINFO
HOMATT2 :: S1.STUDENTACADINFO  ; S2.COURSETAKEN
HOMATT3 :: S1.STUDENTINFO   ; S3.STAFFSTATUS
HOMATT4 :: S2.STUDENTSTATUS ; S3.STAFFSTATUS
```

Figure 5.7. Homonym part of the sample source file.

Each line is read in from the source file. For this case, assume that the first line is read in as shown below:

HOMATT1 :: S1.STUDENTINFO ; S2.COURSETAKEN ; S3.STAFFINFO

Now, as described earlier, this means that the attributes homatt1 of relations STUDENTINFO, COURSETAKEN and STAFFINFO are homonymous, or in other words, these three attributes, though being referred by the same name, need to be identified as three different attributes. The solution involves in generating new names for all the involved attributes, and replacing the original attribute names with the newly generated names.

5.4.1.1 Issues

The solution involves in manipulating the attributes involved in such a way that all the involved attributes (homatt1, homatt1 and homatt1 of the three different relations in this case) would not be identified as the same when a search is done on any one of these

particular attribute (homatt1). A simple solution would be to add the newly generated name as a new entity to each of the attributes involved, in this case, homatt1 of each of the three source relations. But again, problems would arise in the next stage, which is generating the mapping, where in, the updated attribute names need to be readily accessible to generate the intersection and finally the mapping. Hence another solution needs to be arrived at by way of which, the new attribute names would be readily and easily accessible, as would the original names of these attributes if necessary and the mapping between the original and the new names.

5.4.1.2 Implementation

The solution involves in adding the new name for the involved attribute as the third element to vector V3 [figure 5.6] of the attribute in question. One also needs to keep track of the original names of these attributes. The solution involves in adding a new hashtable to the vector V2 (for the specific relation in question), which retains the mapping between the old and the new attribute names. This step is detailed out later in this section. So the steps that are involved in renaming each of the attributes are:

1. Generating new names for each of the attributes involved
2. Adding the new attribute name to the vector V3
3. Adding a new hashtable (if it does not already exist for that particular relation) and adding an entry into that table with the new name of the attribute as the key and the original name as the value

The steps mentioned above are detailed with the example below.

Step1: Generating new attribute names

This step involves generating new names for each of the attributes involves, as all the attributes have the same name, and need to be changed. A method has been written, which takes the current name of the attribute as the input and returns a new generated name as the result. In our example, passing homatt1 of relations STUDENTINFO, COURSETAKEN and STAFFINFO in turn will generate three different names – homatt1_001, homatt1_002 and homatt1_003 respectively.

Step2: Updating vector V3

Figure 5.6 shows the vector V3 as it would look for all the attributes initially, when no new name has been added. Figure 5.8 shows the vector V3 of the attribute homatt1 being updated with the new name homatt1_002 by adding it as the third element to V3.

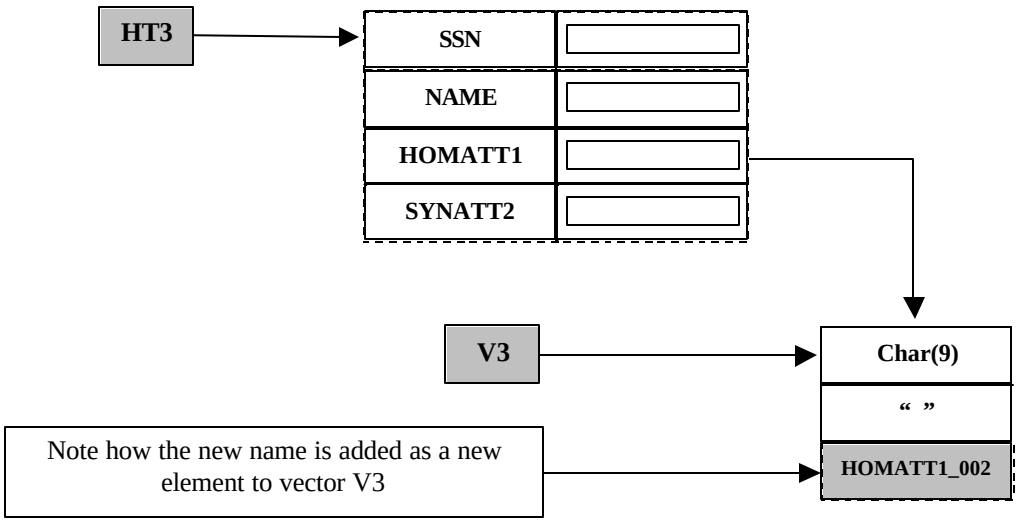

Figure 5.8. Applying homonyms – step 1.

Step3: Adding an entry into hashtable

For each of the attributes that need to be renamed, a method would return the vector V2 for this specific source relation, as each of the source relations will have one such vector.

In this example, as one considers the attribute homatt1, the method will be called with S2 and COURSETAKEN as the parameters for the source and relation names. This method would return the vector V2 for this specific relation COURSETAKEN.

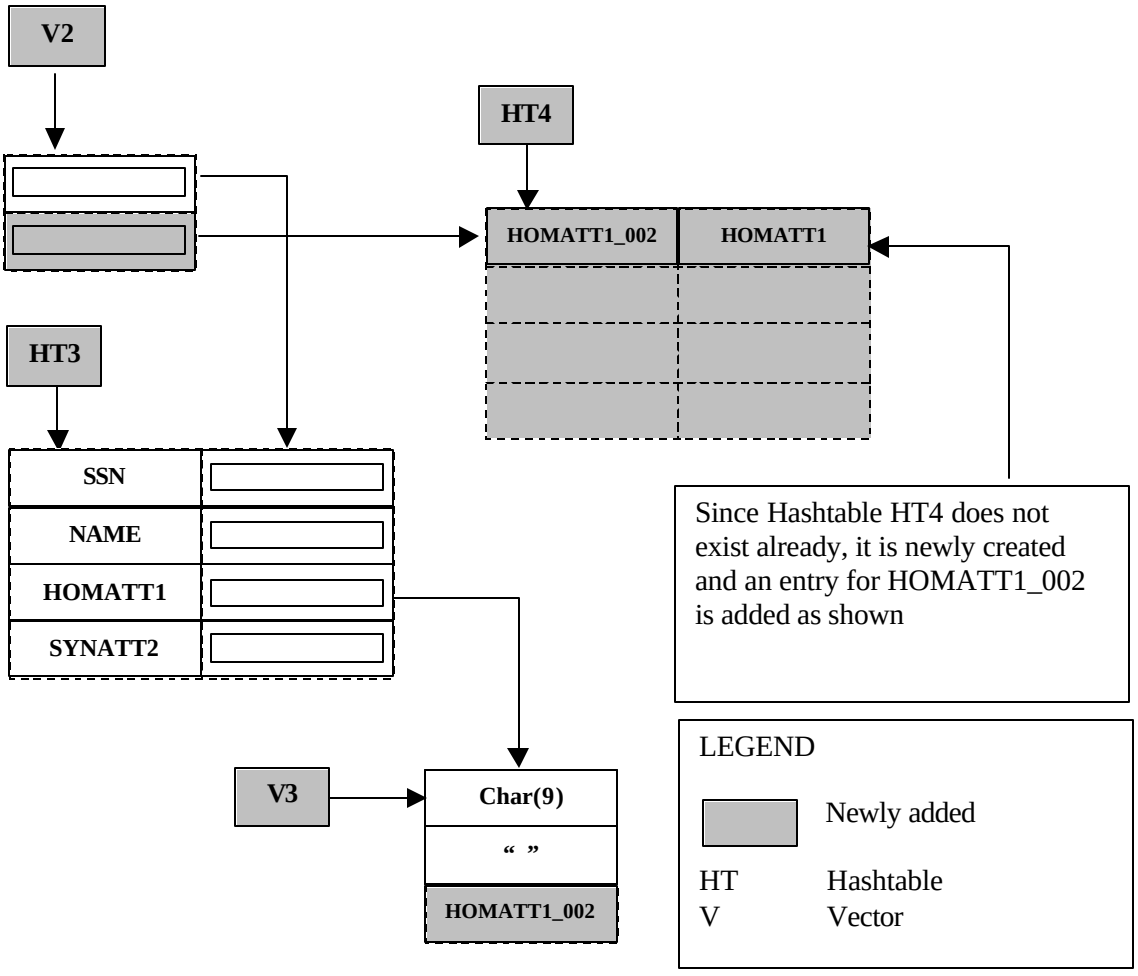

Figure 5.9. Adding homonyms – step 2.

Once vector V2 is obtained, a check is done to see if the hashtable HT4 already exists, as only one such table is created for each of the relations. So while renaming an attribute of a relation that already had one attribute renamed, an initial check is done to see if the table exists. If the hashtable does not exist, then a new element is added to vector V2 and the hashtable HT4 is created. Then a new entry into the hashtable with the new name of the attribute as the key and the old name as the value is added, as shown in figure 5.9. If the hashtable HT4 does exist, then a new entry is added to the existing table, without creating a new one. In our example, the hashtable HT4 does not exist already. Hence the hashtable is created for the source relation COURSETAKEN, and a new entry is added to it. This example demonstrates the renaming process for one attribute, namely homatt1 of source relation COURSETAKEN.

5.4.2 Adding the synonyms

When the synonyms are read from the source file, demarked by the keyword "***synonyms***", the data structure is modified to accommodate these new entries. Again, for easy comprehension, assume a source file, part of which is shown in figure 5.10.

```
synonyms::
S1.STUDENTINFO.SYNATT1 :: S2.COURSETAKEN.SYNATT2 ; S3.STAFFINFO.SYNATT3
S1.STUDENTINFO.SYNATT4 :: S2.STUDENTSTATUS.SYNATT5
S2.COURSETAKEN.SYNATT6 :: S3.STAFFSTATUS.SYNATT7 ;  S1.STUDENTACADINFO.SYNATT8
S2.STUDENTSTATUS.SYNATT9  :: S3.STAFFSTATUS.SYNATT10
```

Figure 5.10. Synonym part of the sample source file.

Each line is read in from the source file. For this case, assume that the first line is read in as shown below:

S1.STUDENTINFO.SYNATT1 :: S2.COURSETAKEN.SYNATT2 ; S3.STAFFINFO.SYNATT3

Now, as described earlier, this means that synatt1, synatt2 and synatt3 of the respective source relations are synonymous, or in other words, these three attributes, though having different names, need to be identified as the same attribute. Hence, the attributes synatt2 and synatt3 are renamed to "synatt1" as per the design, wherein, the first name is retained and the rest of the attributes are transformed with the first attribute's name.

5.4.2.1 Issues

The solution involves in manipulating the attributes involved in such a way that all the involved attributes (synatt1, synatt2 and synatt3 in this case) would be identified when a search is done on this particular attribute (synatt1). A simple solution would be to add the new name as a new entity to each of the attributes involved, in this case, synatt2 and synatt3. But problems would arise in the next stage, which is generating the mapping, where in, the updated attribute names need to be accessed to generate the intersection and finally the mapping. Hence another solution needs to be arrived at by way of which, the new attribute names would be readily and easily accessible, as would the original names of these attributes if necessary and the mapping between the original and the new names.

5.4.2.2 Implementation

The solution here involves two steps, the first being adding the new name for that attribute as the third element to vector V3 [figure 5.6] of the attribute in question. The second step is necessary to make the new names easily accessible and to keep track of the original names of the attributes. This step involves in adding another hashtable to the vector V2 (for the specific relation in question), which retains the mapping between the old and the new

attribute names. This step is detailed out later in this section. So the two steps that are involved in renaming each of the attributes are:

1. Adding the new attribute name to the vector V3
2. Adding a new hashtable (if it does not already exist for that particular relation) and adding an entry into that table with the new name of the attribute as the key and the original name as the value

The two steps mentioned above are detailed with the example below.

Step1: Updating vector V3

Figure 5.6 shows the vector V3 as it would look for all the attributes initially, when no new name has been added. Figure 5.11 shows the vector V3 of the attribute synatt2 being updated with the new name synatt1 by adding it as the third element to V3.

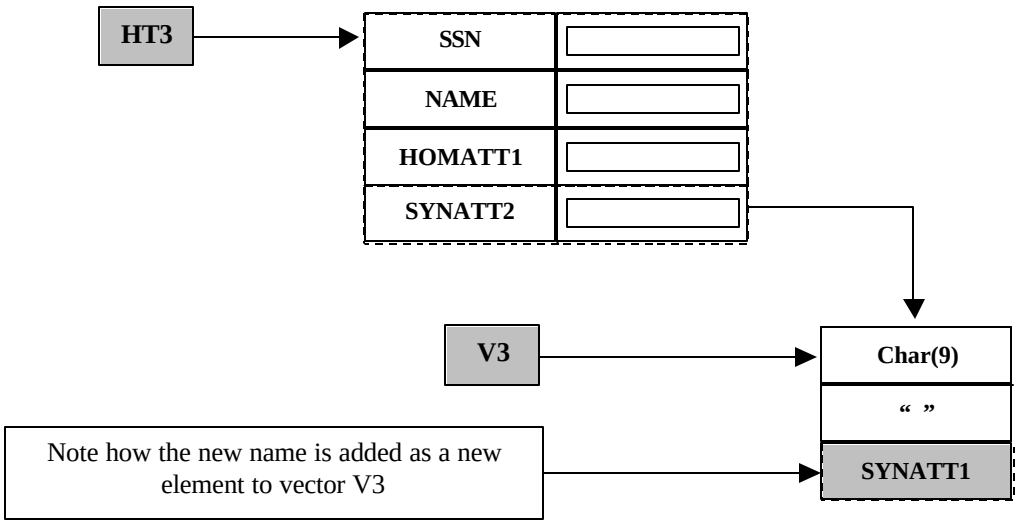

Figure 5.11. Adding synonyms – step 1.

Step2: Adding entry into hashtable

Figure 5.5 shows a vector V2, which initially had only one element. Section 5.3.1 describes the purpose of this vector. Now, for each of the attributes that need to be renamed, a method would return the vector V2 for this specific source relation, as each of the source relations will have one such vector. In this example, if the attribute synatt2 is considered, then the method will be called with S2 and COURSETAKEN as the parameters for the source and relation names. This method would return the vector V2 for this specific relation COURSETAKEN.

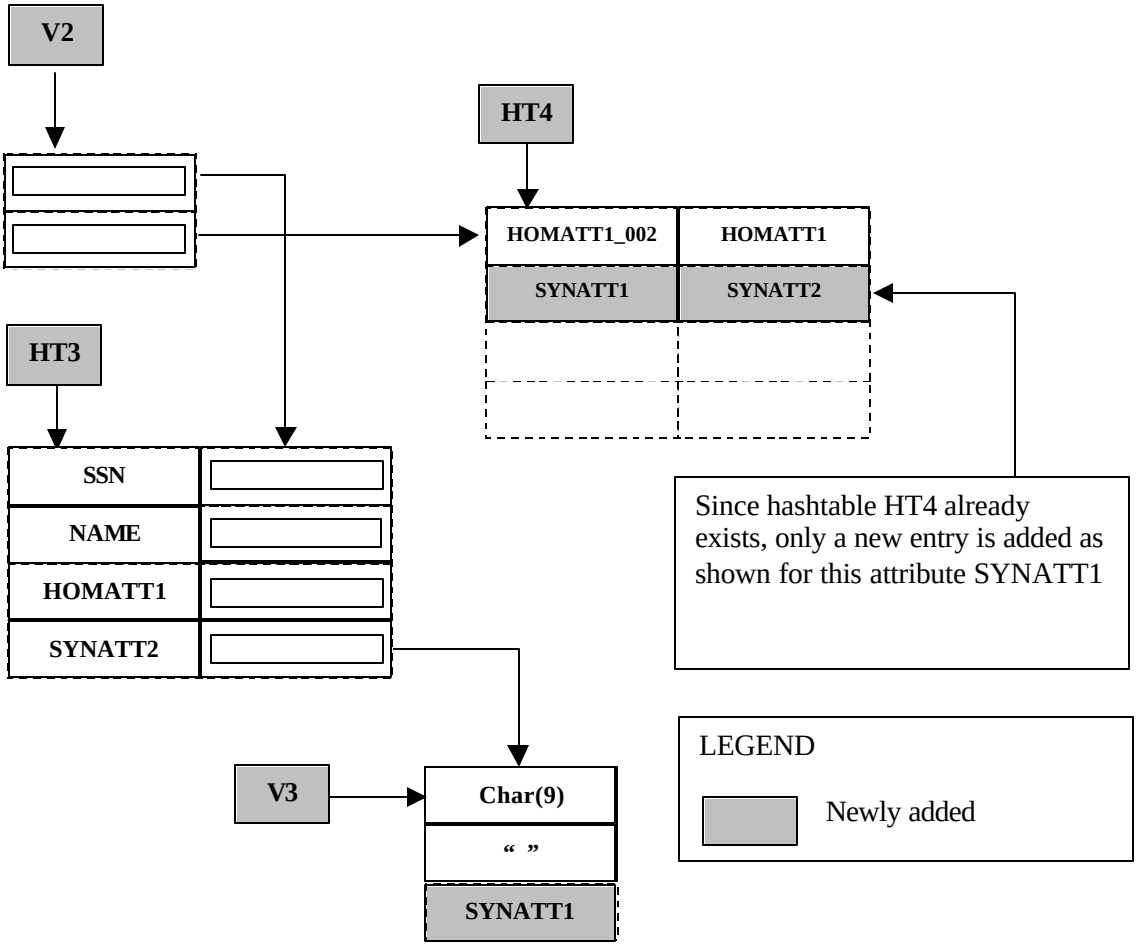

Figure 5.12. Adding synonyms – step 2.

Once vector V2 is obtained, a check is done to see if the hashtable HT4 already exists, as only one such table is created for each of the relations. So while renaming an attribute of a relation that already had one attribute renamed, an initial check is done to see if the table exists. If the hashtable does not exist, then a new element is added to vector V2 and the hashtable HT4 is created. Then a new entry into the hashtable with the new name of the attribute as the key and the old name as the value is added, as shown in figure 5.9. If the hashtable HT4 does exist, then a new entry is added to the existing table, without creating a new one. In this example as shown in figure 5.12, as the hashtable HT4 already exists, one just adds a new entry to it. This example demonstrates the renaming process for one attribute, namely synatt2 of relation COURSETAKEN.

5.4.3 Storing the attribute mappings

The process involved in storing the attribute mappings to the datastructure is no different than the storing of the synonyms and homonyms as detailed out in the previous couple of sections. From section 5.2.4, it is clear that the attribute mappings are defined very much similar to the synonyms, except for the fact that this mapping is between the source and the warehouse schemas, as against two sources. Though it supports three different cases, there is simply a one to one mapping between the DW attribute and the corresponding attribute from the source relation, which simply means that this attribute of this warehouse relation is the same as the attribute of the given source relation, the only difference being the change of name by which it is referred by.

Again, the steps involved here are the same as that covered under section 5.4.2 for storing the synonyms, differing only by the relations involved, warehouse relations in this case against source relations in the former case (synonyms).

5.4.4 Storing the derived attributes

As mentioned in chapter 3, the derived attributes need to be handled separately. They cannot be added to the data structure until after the generation of the mappings, as they are not required to generate the same. As they are not required until after the generation of the mappings and user validation, the derived attributes and the deriving attributes from the source relations are stored in a temporary store initially, and would be covered again after the generation of the mappings for all the warehouse relations.

5.4.5 Completing the hashtable

From sections 5.4.1, 5.4.2, 5.4.3, it is bare that for all attributes of the source and the warehouse relations that require to be referred to by a different name (under normalization), an entry is created in a new hashtable HT4 which would contain the *<new name, old name>* pair for each of the attributes. But at the end of the normalization process, those attributes that do not require a name change, or in other words, those attributes that have not been transformed are not accounted for in this new hashtable HT4. The problem arises in the later stage when this hashtable is the only structure referred to, to get the set of attributes for the various relations to generate the possible mappings.

5.4.5.1 Implementation

The implementation is a simple on which involves in obtaining a set of attributes for each of the source and warehouse relations that are not present in the hashtable HT4 for the same, and adding an *<original name, original name>* pair to the hashtable for each of the remaining attributes. Figure 5.12 shows the incomplete hashtable HT4 that would be completed as shown in figure 5.13.

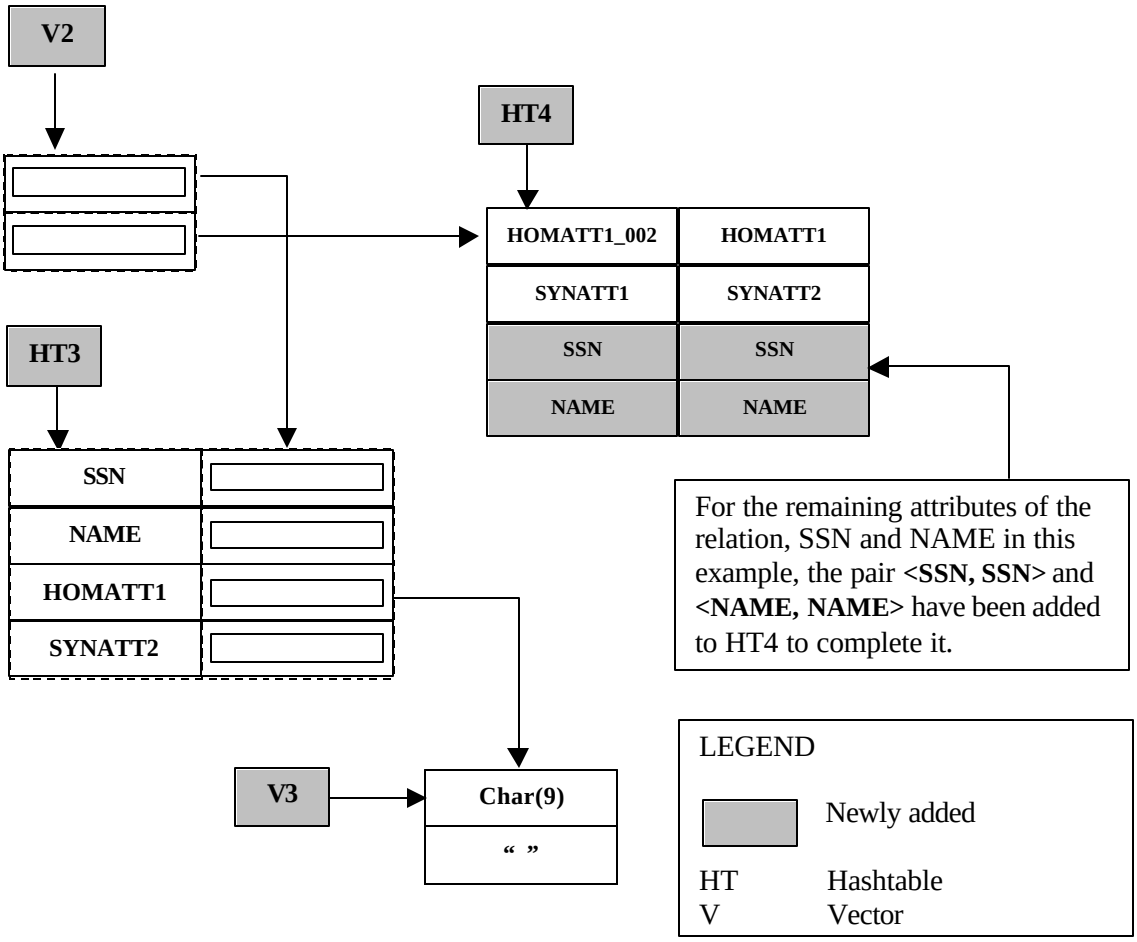

Figure 5.13. Completing the hashtable HT4.

5.5 Intersection

As mentioned earlier, this stage involves generating a list of possible source relations that have at least one attribute in common with the warehouse relation in question, along with the set of attributes that the relations have in common with this warehouse relation. This information is stored in a structure, the description of whose implementation follows in this section. At the end of this stage, one is required to have the following information for each of the warehouse relations in question namely,

1. The list of source relations that have at least one attribute in common with this warehouse relation in question.
2. For each of these source relations, a set of attributes that these relations have in common with the warehouse relation.

The implementation would be best explained with an example. This would be referred back in both the remaining parts of the algorithm. Let us assume a single warehouse relation. The same would be repeated for each of the rest of the warehouse relations. Assume the following information for the warehouse relation as described in figure 5.14 and table 5.1. One needs to note here that the source relations and the attributes are selected in such a way that they cover all the possible scenarios.

Warehouse Name: **DW1**
Warehouse Relation Name: **DW-R1**
Attribute Set: [**A B C D E F**]

Also assume that the following source relations have some attributes in common with this warehouse relation as shown below (note that all the common attributes are highlighted):

Figure 5.14. Example to demonstrate the matching algorithm.

Table 5.1. Example schemas.

Source	Relation	Attribute Set
S1	R1	{**A B C D** X Y Z}
S1	R2	{**D E F** M N}
S2	R3	{**A B C D E F**}
S2	R4	{**A B C D E F**}

5.5.1 Issues

This section covers the various issues involved in implementing this part of the algorithm.

1. To enable proper comparisons of the attributes, one is required to consider the **transformed** [refer chapter 3, section 3.5] attribute names for the purpose of comparison.
2. As the pseudo-code of the intersection part of the algorithm in chapter 3 illustrates, for each of the warehouse relations, one is required to do an attribute-attribute comparison between the attributes of the warehouse relation and the attributes of the various source relations. The problem involves in arriving at some way to improve the performance of this attribute-attribute comparison, which otherwise would require one to do the comparison N x M times [N being the total number of attributes of all the source relations and M the number of attributes of the warehouse relation in question].
3. For each of the warehouse relations, a way has to be devised by which the result of the comparisons, which is the list of source relations that have any attribute in common with the warehouse relation and the set of common attributes for each of the source relations in question can be stored somewhere for easy retrieval in the next stage.

5.5.2 Implementation

Sections 5.4.1, 5.4.2 and 5.4.3 detail out part of the transformation process, wherein the transformed attribute names are stored in a hashtable HT4 as illustrated in figure 5.13. This hashtable is completed as described in section 5.3.6. One needs to note at this point that this hashtable HT4 with the <new, old> attribute name pairs would exist for all the relations

of the schemas, including the warehouse relations. Hence getting back a set of transformed attributes for any of the relations doesn't seem to pose a problem anymore.

The other main issue involved here is the attribute comparison between the source and the warehouse relations. To improve the comparison from a naïve N x M number of comparisons, [N being the total number of attributes of all the source relations and M the number of attributes of the warehouse relation in question] a method has been formulated that would decrease the number of comparisons from N x M to just N. For each of the warehouse relations, instead of obtaining the set of normalized attributes, the whole hashtable HT4 is obtained. For each of the source relations, the set of normalized attributes is obtained by returning just the key set of the hashtable HT4 for that source relation in question. Now, comparison involves in checking if each of the attribute of the source relations exist in the warehouse relation, by hashing into the hashtable HT4 for the warehouse relation. This way, the performance of the comparison is tremendously improved from N x M to just N.

One still needs some way to store the list of source relations and the common attributes for each of the warehouse relations. To implement this, a new data structure "**intersectVector**" is created, which, will contain the following information for each of the warehouse relations at the end of this stage:

1. The warehouse name.
2. The warehouse relation name (of the relation in question).
3. The number of source relations that it has at least one attribute in common with. One needs to note here that it will be updated as and when a new source relation with some common attribute with the warehouse relation is found.
4. A list of the source relations that have common attributes with this warehouse relation. Again, for each of these source relations added to this structure, the set of common attributes of that relation are added too.

Section 4.3 of chapter 4 details out this data structure and figure 4.2 illustrates the same. The storing of data in this structure is best described with the given example [figure 5.14]. The structure "intersectVector" is shown in figure 5.15 when it is first created. It would consist of only the main vector V1.

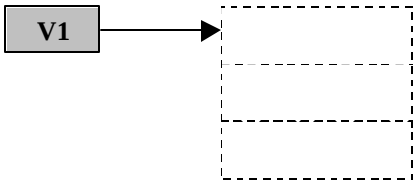

Figure 5.15. Initial intersect vector.

Now, as per the example, an instance for the warehouse relation "DW-R1" is created, where by, vector V2 is added as an element to vector V1 and the first three elements of this new vector V2 would contain the warehouse name, the warehouse relation name, and a field that gives the number of source relations that have attributes in common with this warehouse relation (initialized to ZERO) respectively, as shown in figure 5.16.

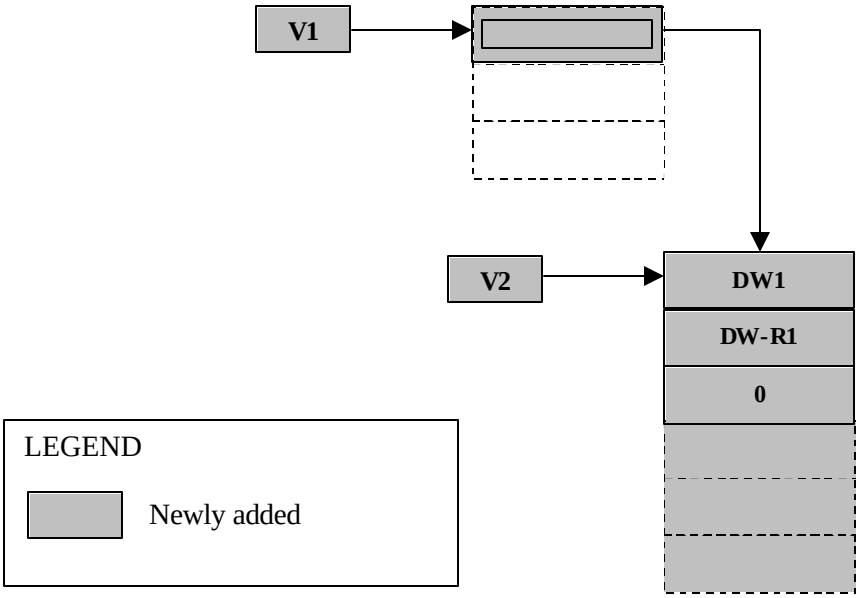

Figure 5.16. Intersect vector – after creating an entry for the first warehouse relation.

As explained earlier, for each of the source relations, the set of attributes is retrieved from the stored data structure. Each attribute of each of the source relation is considered and compared with the hashtable HT4 of the warehouse relation (DW-R1 in this case), by hashing the attribute name into the hashtable to check if that attribute exists. If it exists, then that particular attribute is added to a temporary set. Once all the attributes of a source relation have been checked for existence with the warehouse relation, this temporary set of attributes along with the source name and the source relation name are added to the datastructure "intersectVector" by adding a new entry into vector V2, which would be a structure (attribSet) which has exactly three elements – one for the source name, one for the source relation name and the third, a set (for the attribute set).

This is illustrated in figure 5.17 for the first source relation, namely R1 of source S1 as per the example. In the example one can see that this relation R1 has the attributes [A B C

D] in common with the warehouse relation DW-R1. One thing to note here is that if a source relation has no attribute in common with the current warehouse relation, it is simply ignored.

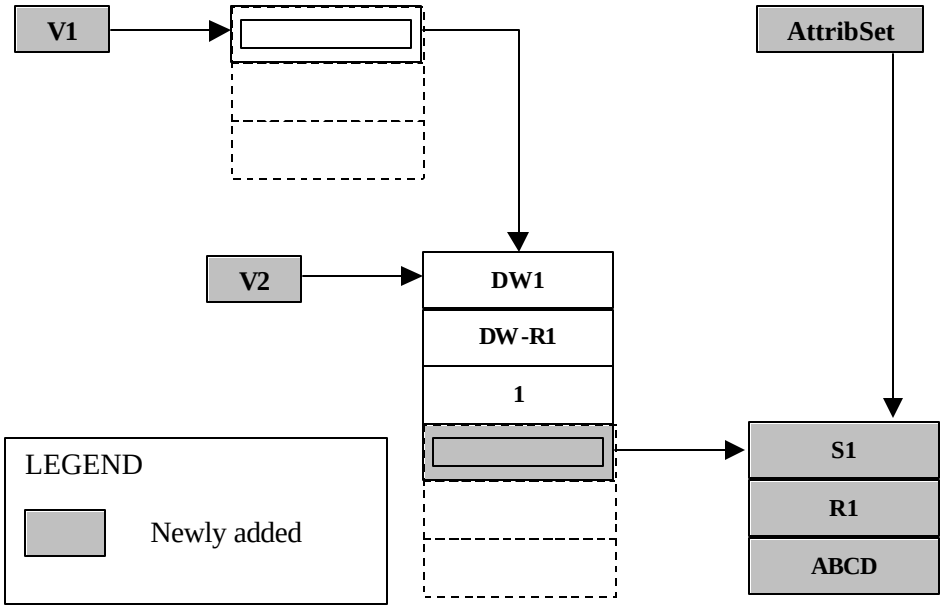

Figure 5.17. Intersect vector – after adding the first source relation.

Again, the same steps are followed for each of the remaining source relations until all the relations have been compared against this warehouse relation in question. Then the whole cycle is repeated for the rest of the warehouse relations. The structure "intersectVector" after comparing the attributes of the warehouse relation DW-R1 with the source relations of the given example is as shown in figure 5.18.

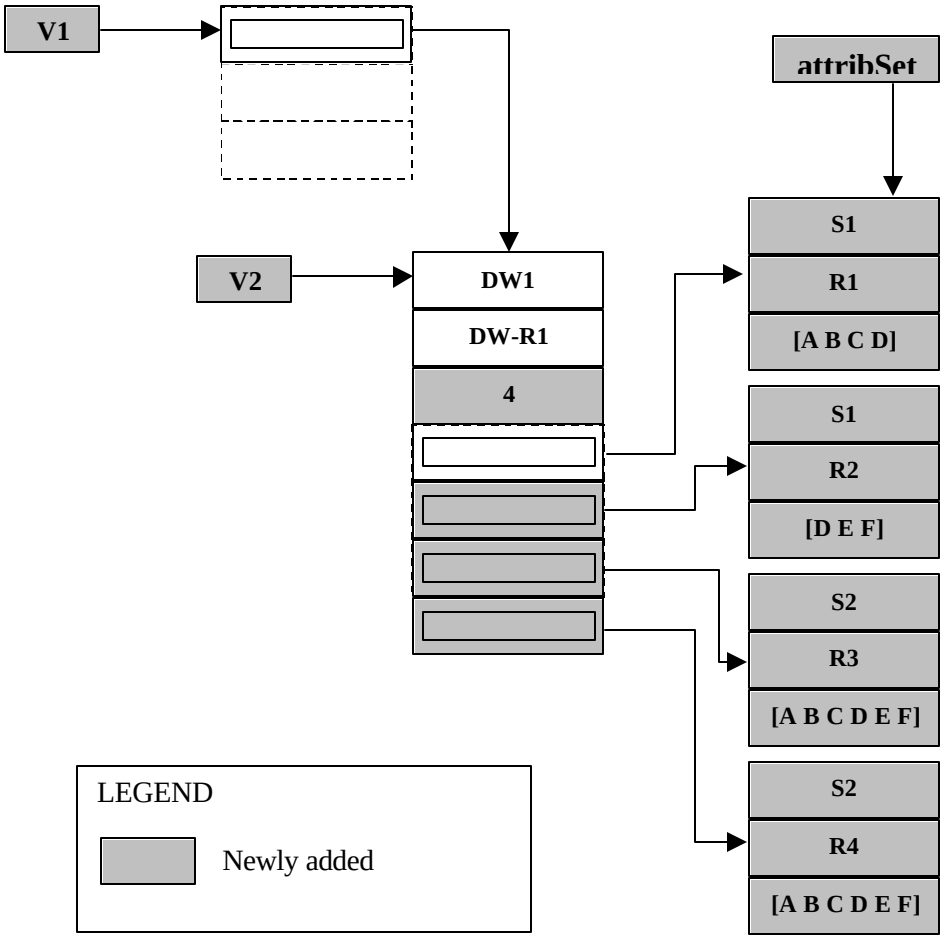

Figure 5.18. Intersect vector – at the end of intersection.

5.6 Mapping generation

Once the set of source relations that have some attribute(s) in common with the warehouse relation is obtained and stored in the structure "intersectVector" as detailed out in the previous section, the next stage would be to generate all mappings that can possibly result from the source relations to the warehouse relation in question. One also needs to figure out the kind of the mapping – whether it is a projection, join, or a union/ intersection. The remaining sections describe the implementation of this final stage of the matching algorithm.

The resulting structure "**intersectVector**" from the previous stage, that stores the set of intersecting attributes and the corresponding source and relation names for each of the data warehouse relation is the only structure that is required in this stage. As mentioned in chapter 3, the quest here is to identify how each of the warehouse relations is derived from the source relations. The various possible scenarios are:

1. A projection from a single source relation (maybe complete or partial)
2. A join of two or more source relations (it maybe a normal join or a cartesian join, and a complete or a partial projection on join)
3. A union or intersection of two or more source relations (again, it may be a complete or partial projection on union)

For each of the warehouse relations**,** the sets of attributes of the source relations are retrieved and analyzed and the plausible mappings generated. The various stages involved are:

1. Check for any projection
2. Check for a join, union and intersection

Again, at the end of this stage, one is required to have the following information for each of the warehouse relations in question namely,

1. The various mappings that can be possible between the source relations and the warehouse relations, giving the various possible mappings of the source relations that make up a projection, join or a union / intersection.

5.6.1 Issues

One issue that is involved in the implementation of this stage of the matching algorithm is the storing of the generated mapping information for each of the warehouse relations. One needs the following information to be stored for each of the generated mappings for any warehouse relation:

1. The type of mapping- whether a projection, join, or union / intersection
2. In any case, one needs to store the source name, the relation name and the set of attributes that the particular source relation has in common with the warehouse relation.

5.6.2 Implementation

A third and final structure named "checkVector" is created for the sole purpose of storing the information of the generated mappings for each of the warehouse relations. There would exist an instance of this structure for each of the warehouse relations. This structure has been detailed out in section 4.4, chapter 4. The next couple of sections describe the two steps involved in this stage namely check for projection and check for join.

5.6.3 Check for projection

For each of the source relations from the structure "**intersectVector**", the set of common attributes is compared with the set of attributes of the warehouse relation (excluding the derived attributes if any) in question for equality. If the result of the check is true, it implies that the warehouse relation is indeed a projection of that source relation. This check is not stopped when a match is found, and is done for all the source relation entries in the structure "intersectVector" that warehouse relation in question, as one might be able to derive the warehouse relation from more than one source relation. For every check where the result is true, the source relation information is added to the new data structure "**checkVector**". A new vector V2 is added as a new element to the existing vector V1, and this in turn would have two elements, one for the type of projection (complete or partial) and the other to store the "attribSet" structure of the source relation in question, which would contain all the required information about the source relation, namely, the source name, the source relation

name and the set of common attributes. Again, this is best illustrated with an example. The initial empty "checkVector" is as shown in figure 5.19.

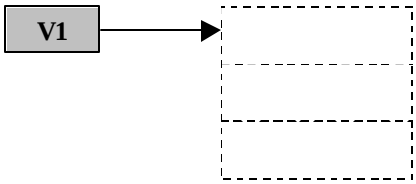

Figure 5.19. Initial check vector.

In the given example [figure 5.18], relations R4 and R5 of source S2 would satisfy this check condition, as there is a possibility of the warehouse relation DW-R1 being a partial projection of these relations. The checkVector after this information being added to it is as illustrated in figure 5.20.

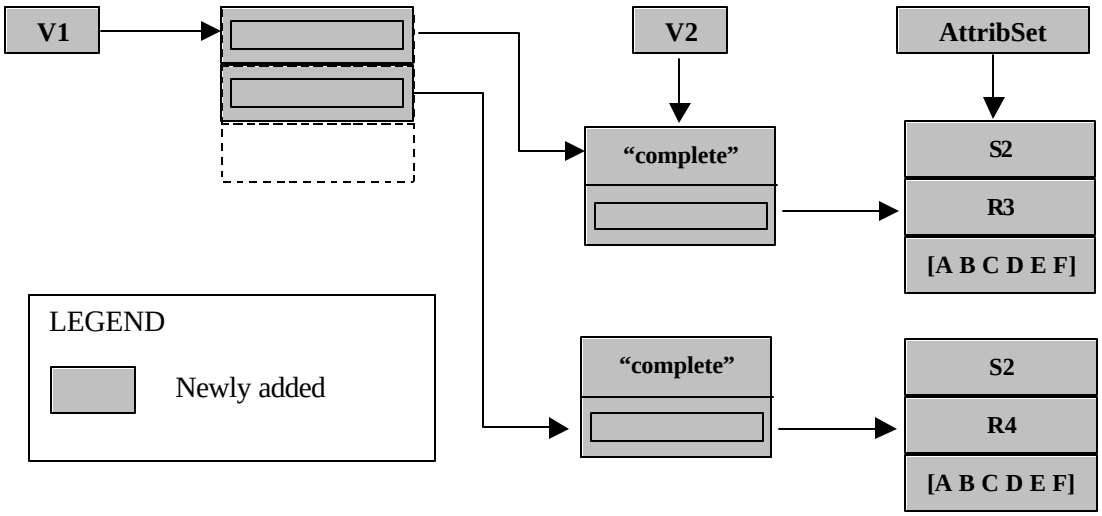

Figure 5.20. Check vector after projection.

A new vector V2 is added as a new entry to vector V1 and the attribSet structure of relations R4 and R5 are added to V2 as shown in figure 5.20.

5.6.4 Check for join, union and intersection

This check is done only if the no of source relation entries in "**intersectVector**" is greater than ONE, as it takes at the least two relations to make up a join or a union. Now, again, for all of the source relations that are listed in the data structure "**intersectVector**" for the warehouse relation in question, all possible combinations of source relations are considered, and the combined set of common attributes of each pair of source relations are obtained and checked with the warehouse relation for equality. Join and union/ intersection are determined by the following checks.

5.6.4.1 Check for join

Whenever there is a match between the combined attribute set of the source relation pair and the warehouse relation, it implies that this join of source relations forms the warehouse relation, and that it is a valid mapping. This mapping information is added to the data structure "**checkVector**". The things to figure out here are

1. The kind of join – a join with a common attribute (usually the key attribute), or a Cartesian join, with no attributes in common
2. Whether it is a complete or a partial projection on join

Again, a new vector V2 is added as a new element to the existing vector V1, and this in turn would have three elements instead of two as in projection, one for type of join (complete or partial projection on join) and the remaining two to store the "attribSet" structures of the two source relations that make up the join of the warehouse relation in question, which would contain all the required information about the source relations, namely, the source name, the source relation name and the set of common attributes. Again, this is best illustrated with an example. From the given example [figure 5.18], it is obvious that a number of joins can be possibly generated to derive the warehouse relation as illustrated in table 5.2.

Table 5.2. All possible joins.

JOIN OF	JOIN ATTRIBUTES	TYPE
S1.R1 U S1.R2	[D]	PARTIAL
S1.R1 U S2.R3	[A B C D]	PARTIAL
S1.R1 U S2.R4	[A B C D]	PARTIAL
S1.R2 U S2.R3	[D E F]	PARTIAL
S1.R2 U S2.R4	[D E F]	PARTIAL
S2.R3 U S2.R4	[A B C D E F]	COMPLETE

All of these possible mappings are added to the structure checkVector. But for the purpose of illustration, only the first mapping is added as illustrated in figure 5.21. This completes this stage wherein the various plausible mappings in the form of joins are generated for each of the warehouse relations.

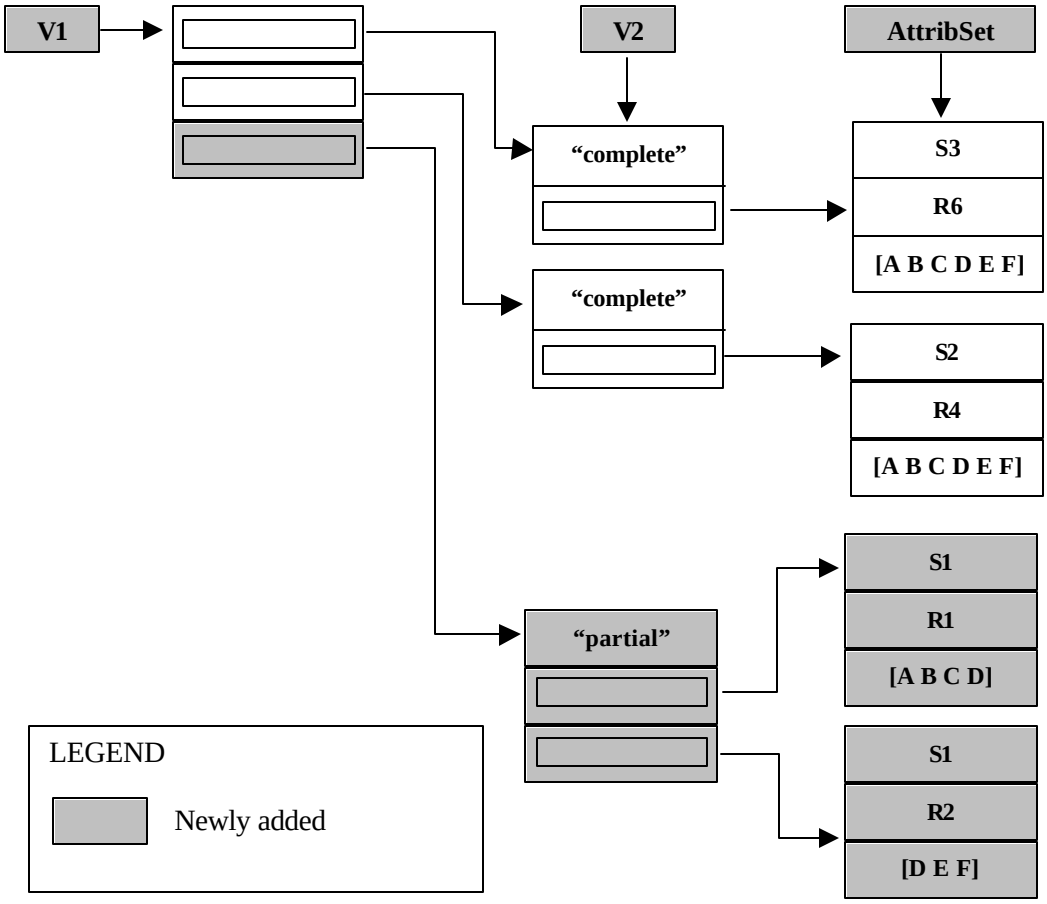

Figure 5.21. Check vector after join.

5.6.4.2 Check for union or intersection

Whenever there is a match between the combined attribute set of the source relation pair and the warehouse relation, and also, if the common attributes of the two source relations of that pair turn out to be equal, then it implies that this is a case of union or intersection of the source relations forming that particular pair. This mapping information is added to the data structure "**checkVector**". The thing to figure out here is the kind of union / intersection – Whether it is a union on a complete or a partial projection of the source relations.

Again, for each valid mapping, a new vector V2 is added as a new element to the existing vector V1, and this in turn would have three elements as in join, one for type of union / intersection (complete or partial projection) and the remaining two to store the "attribSet" structures of the two source relations that make up the union / intersection of the warehouse relation in question, which would contain all the required information about the source relations, namely, the source name, the source relation name and the set of common attributes. One needs to note here that any one pair of source relations that satisfy the above checks can either be a union or an intersection, which can only be determined by examining the tuples of each of the source relation in question as well as those of the warehouse relation. As that is not done at this point, where all the relations are handled at a relational level, it is left as is for now. Again, this is best illustrated with an example. From the given example [figure 5.18], it is obvious that the only possible union or intersection is the one of relations R3 and R4 of source S2.

All of these possible mappings are added to the structure checkVector as illustrated in figure 5.22. This completes this stage wherein the various plausible mappings in the form of unions or intersections are generated for each of the warehouse relations.

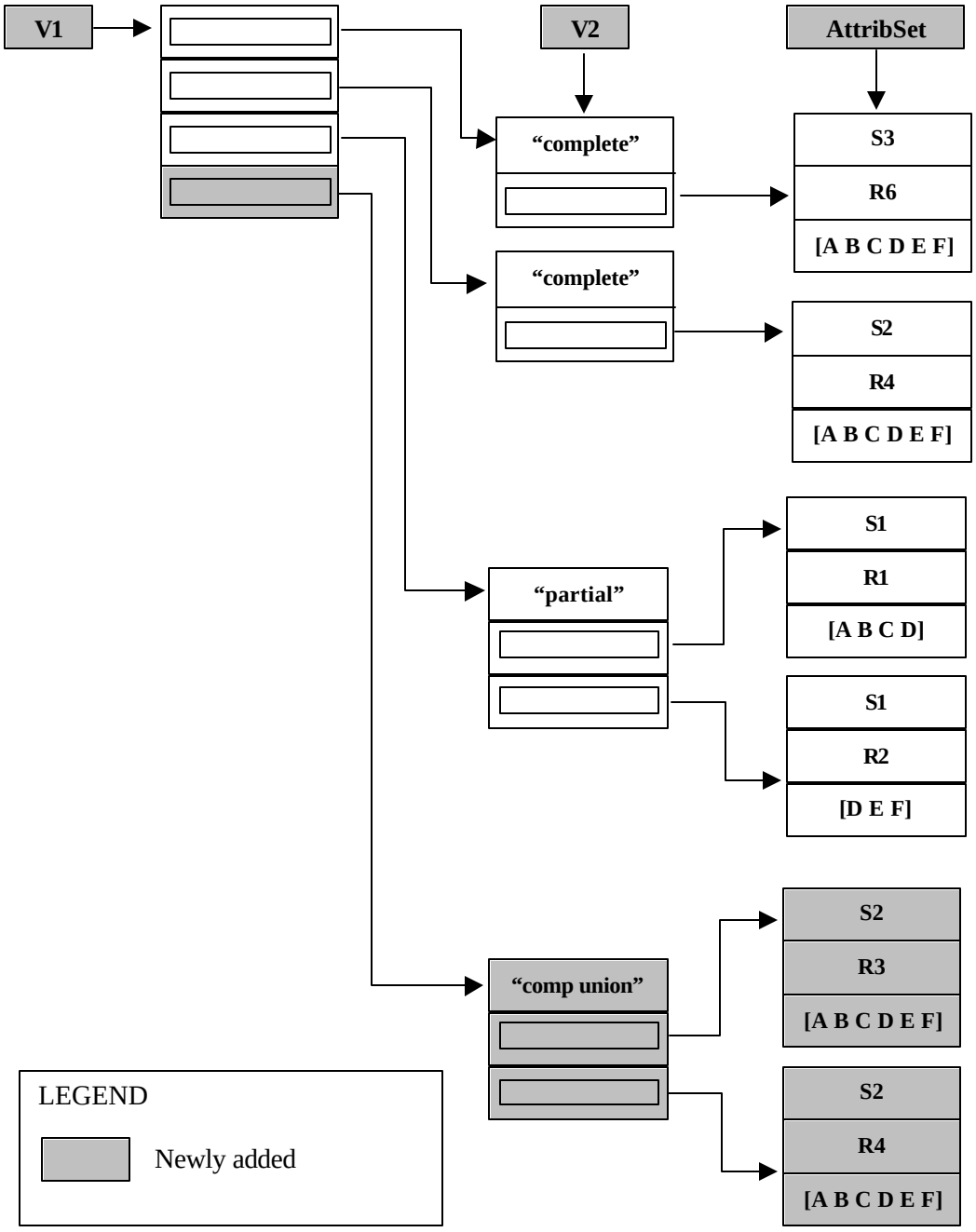

Figure 5.22. Check vector after union / intersection.

5.7 User validation

The final step in the design involves the intervention of the user, where the system spews out all the possible mappings that it has generated for each of the warehouse relations and waits for the user to validate the results with the one mapping that would be appropriate for the case as required by the user. One needs to note here again that as already illustrated in the previous chapters, this is not a completely automated process, and still requires the user validation at the end to obtain the final exact mapping for the warehouse relation as required or envisioned by the user at an earlier stage. Once this validation is done for all the warehouse relations in question, the next step would be the generation of the triggers for setting up the warehouse for updates. This part is left as part of future work that would be covered in the following chapter.

5.8 Chapter summary

This chapter covered the implementation of the matching algorithm and the issues involved in the same. The next chapter talks about the performance optimizations done on the algorithm.

CHAPTER 6
PERFORMANCE OPTIMIZATION AND TESTING

6.1 Overview

This chapter evaluates the performance of the matching algorithm and presents one with the various optimization techniques that have been implemented to improve the performance of the algorithm, and also presents other techniques that can be implemented to further improve the performance of the same. It also describes the various tests that have been performed on the algorithm to check for consistency and exactness.

6.2 Implemented optimization techniques

6.2.1 Use of hashtables

An un-optimized code would take an extended time period for searching for attributes or a set of attributes, which have been extensively used in the algorithm.

Optimization here involves in implementing the source and the warehouse relations as hashtables, which facilitates a more easier and effective search for the attributes of the relations, be it source or warehouse.

6.2.2 Reduced number of cycles

In any single run of the intersection phase of the algorithm [section 3.5, chapter 3], given the number of attributes of the source relation being M and the number of attributes of the warehouse relation being N, an un-optimized code would have to run MxN cycles to facilitate the comparison of each of the attributes of the source relation with all the attributes of the warehouse relation.

Optimization here involves in implementing the source and the warehouse relations as Hashtables as mentioned in the previous section, which enables hashing an attribute into the hashtable to check for existence as against making a run through all the attributes of the warehouse relation. This optimization reduces the number of cycles from MxN as described above to just M, which would be just the number of attributes of the source relation for each run. This is illustrated in figure 6.1.

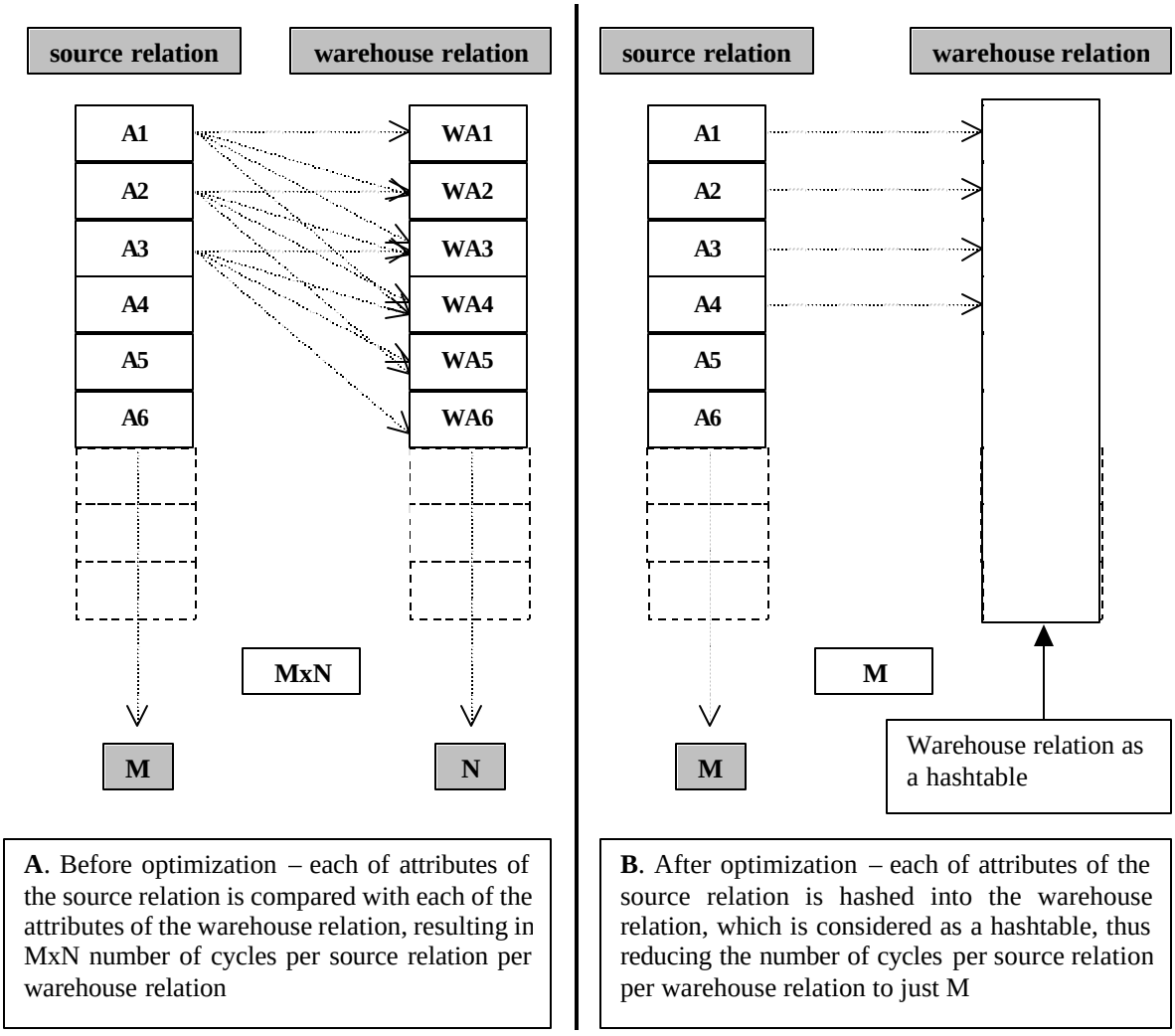

Figure 6.1. Reducing the number of cycles.

Figure 6.1 illustrates a single cycle of the algorithm for the intersection phase that is the main phase that needs optimization. As explained in the previous chapters, in this phase of the algorithm, for each of the warehouse relations, the attributes of each of the source relations need to be compared with the attributes of the warehouse relation to check for equality. As illustrated in the figure, without optimization, this would result in MxN number of cycles to be performed for each source, warehouse relation pair. On optimization, this reduces to just M, which is the number of attributes of the source relation in question, as illustrated in part B of the figure.

6.2.3 Filtered source relations

After the second stage, which is intersection, the list of source relations is effectively filtered, and one ends up with only the list of source relations that have some attribute in common with each of the warehouse relations. This again reduces the total number of comparisons that are required in the subsequent stages.

6.3 Techniques to improve optimization

6.3.1 Parallelization

The algorithm has been implemented in such a way that each cycle involving each one of the warehouse relations can be executed independent of the other. This paves a way for the algorithm to be further optimized by means of parallelization where in the algorithm can be run in parallel for all the involved warehouse relations. Though not implemented, this can be implemented to further improve the performance. This is illustrated in figure 6.2.

When one refers back to the structure "**intersectVector**" [sections 4.3 & 5.7] and the proceedings of the algorithm, it is clear that for each cycle involving each of the warehouse relations, the same steps are executed. One would note at this juncture that each of the cycles have been implemented in such a way that they are totally independent of each other. Hence,

to further optimize the process, one can easily implement parallelization here, wherein, each of the cycles can be made to perform in parallel, without obstructing any of the other cycles. Threads can be safely spawned to perform the same.

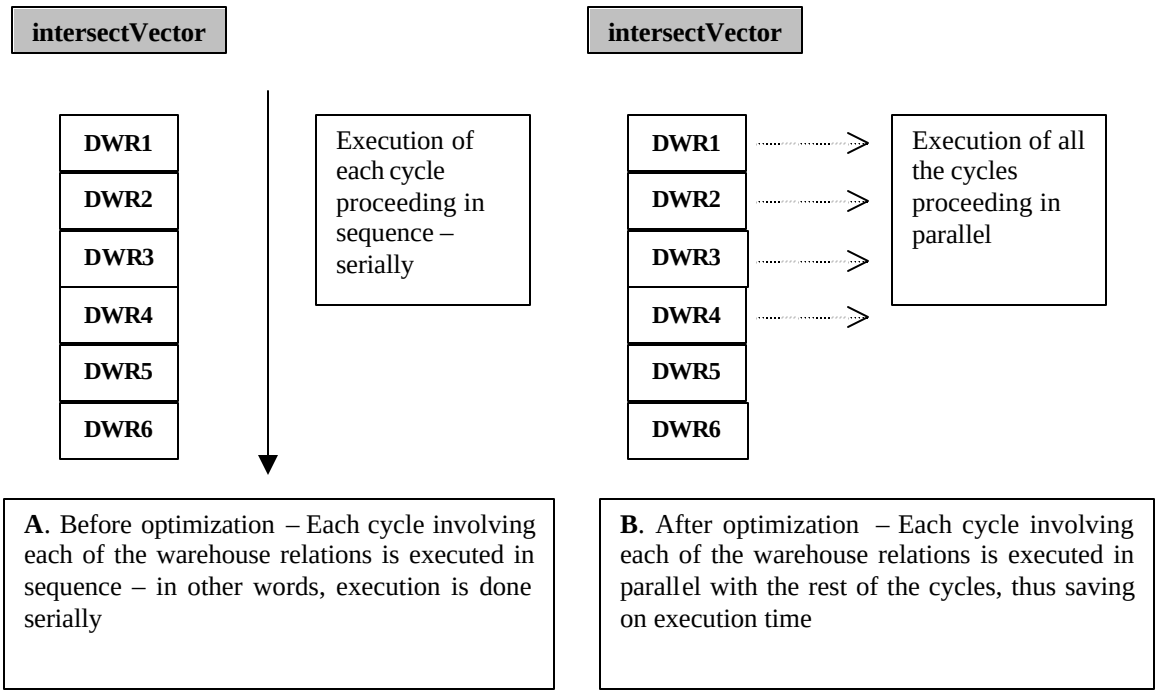

Figure 6.2. Parallelization.

6.4 Testing & test cases

A comprehensive and exhaustive testing has been done on the system to check for consistency and correctness, ranging from checks for boundary conditions to checks to ensure that each part and phase of the algorithm performs as claimed. The list of the tests performed and the results from the system have been included as part of Appendix A.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

7.1 Summary

To summarize, this thesis addresses the problem of schema matching and the need for automating mapping generation. The complexities involved in automation were also addressed in detail. This thesis designed and implemented a solution for automating schema matching and mapping generation pertaining to data warehouses for the relational domain. This thesis designed and presented a new matching algorithm, and talked about the performance and implementation issues involved. This thesis also looked at improving the performance of the algorithm in various ways, including parallelization.

In conclusion, one needs to realize that this is not the final version of the tool. The tool that has been developed in this thesis:

- Enables understanding of the mapping space
- Enables evaluation of mappings from different viewpoints (Ease of implementation, etc)
- Enables the warehouse designer to choose the appropriate mapping for the warehouse schemas
- Allows warehouse designers to explore several mappings before finalizing the dw schemaThis tool can be extended to include generation of triggers and code for propagating the updates from the source schemas to the warehouse schemas, which has been left as part of future work.

7.2 Future work

Future work on this thesis can involve the following areas as described in the following sub-sections.

7.2.1 Implementing a data dictionary/ thesaurus

This would involve integrating the algorithm with a data dictionary and a thesaurus that would effectively reduce the amount of work that needs to be performed on the part of the user, who at present, has to provide the system with the list of synonyms that would be found in the schemas. Adding a data dictionary and/or a thesaurus would facilitate the system to obtain the set of similar attributes which would be comparable to any given attribute of the schemas in question, which would enable the same to generate a more comprehensive list of mappings between source and warehouse relations, with little user intervention.

7.2.2 Integrating triggers and updates

This would involve extending the system to include the setting of triggers to effectively update the warehouse relations as the source relations are modified by means of CREATE, DELETE or UPDATE. Updating the warehouse relations by means of a difference-based approach as against the trigger-based approach can also be implemented as an alternative.

7.2.3 Extending the system to multiple platforms

Once triggers and updates are integrated with the system as described in the previous section, the system can be extended to support multiple RDBMs, wherein, the system should be able to update the warehouse, irrespective of the platform of the source and warehouse relations. It should be able to support updates across multiple platforms.

APPENDIX A

TEST CASES AND PROGRAM OUTPUTS

Case 01: complete projection

schemas::

source S1 true ORACLEOMEGA

relation R1

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)

relation R2

attribute E char(9)
attribute F char(9)
attribute G char(9)
attribute H char(9)

warehouse DW

relation R

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)

Output 01:

Warehouse Relation: R

no of possible Mappings -- 1

Mapping No:: 1

a single source relation projection
And it is a complete projection of the source relation

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
D
C
B

Case 02: partial projection

schemas::

source S1 true ORACLEOMEGA

relation R1

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)

relation R2

attribute D char(9)
attribute E char(9)
attribute F char(9)

warehouse DW

relation R

attribute A char(9)
attribute B char(9)
attribute C char(9)

Output 02:

Warehouse Relation: R

no of possible Mappings -- 1

Mapping No:: 1

a single source relation projection
And it is a partial projection of the source relation

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
C
B

Case 03: cartesian join (complete projection on join)

schemas::

source S1 true ORACLEOMEGA

relation R1

attribute A char(9)
attribute B char(9)
attribute C char(9)

relation R2

attribute D char(9)
attribute E char(9)
attribute F char(9)

warehouse DW

relation R

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)
attribute E char(9)
attribute F char(9)

Output 03:

Warehouse Relation: R

no of possible Mappings -- 1

Mapping No:: 1

join of two source relations
And it is a complete projection on join

Cartesian join with no join attribute

Source Name -- S1
Relation Name -- R2

Common Attributes::

F
E
D

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
C
B

Case 04: cartesian join (partial projection on join)

schemas::

source S1 true ORACLEOMEGA

relation R1

attribute A char(9)
attribute B char(9)
attribute C char(9)

relation R2

attribute D char(9)
attribute E char(9)
attribute F char(9)

warehouse DW

relation R

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)

Output 04:

Warehouse Relation: R

no of possible Mappings -- 1

Mapping No:: 1

join of two source relations
And it is a partial projection on join

Cartesian join with no join attribute

Source Name -- S1
Relation Name -- R2

Common Attributes::

D

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
C
B

Case 05: join (single join attribute & complete projection on join)

schemas::

source S1 true ORACLEOMEGA

relation R1

attribute A char(9)
attribute B char(9)
attribute C char(9)

relation R2

attribute C char(9)
attribute D char(9)
attribute E char(9)

warehouse DW

relation R

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)
attribute E char(9)

Output 05:

Warehouse Relation: R

no of possible Mappings -- 1

Mapping No:: 1

join of two source relations
And it is a complete projection on join

Equi join with join attributes –
Join Attribute(s)::

C

Source Name -- S1
Relation Name -- R2

Common Attributes::

E
D
C

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
C
B

Case 06: join (single join attribute & partial projection on join)

schemas::

source S1 true ORACLEOMEGA

relation R1

attribute A char(9)
attribute B char(9)
attribute C char(9)

relation R2

attribute C char(9)
attribute D char(9)
attribute E char(9)

warehouse DW

relation R

attribute A char(9)
attribute B char(9)

attribute C char(9)
attribute D char(9)

Output 06:

Warehouse Relation: R

no of possible Mappings -- 1

Mapping No:: 1

join of two source relations
And it is a partial projection on join

Equi join with join attributes --
Join Attribute(s)::

C

Source Name -- S1
Relation Name -- R2

Common Attributes::

D
C

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
C
B

Case 07: join (multiple join attributes & complete projection on join)

schemas::

source S1 true ORACLEOMEGA

relation R1

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)

relation R2

attribute C char(9)
attribute D char(9)
attribute E char(9)
attribute F char(9)

warehouse DW

relation R

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)
attribute E char(9)
attribute F char(9)

Output 07:

Warehouse Relation: R

no of possible Mappings -- 1

Mapping No:: 1

join of two source relations
And it is a complete projection on join

Equi join with join attributes --
Join Attribute(s)::

D
C

Source Name -- S1

Relation Name -- R2

Common Attributes::

F
E
D
C

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
D
C
B

Case 08: join (multiple join attributes & partial projection on join)

schemas::

source S1 true ORACLEOMEGA

relation R1

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)

relation R2

attribute C char(9)
attribute D char(9)
attribute E char(9)
attribute F char(9)

warehouse DW

relation R

attribute B char(9)
attribute C char(9)
attribute D char(9)
attribute E char(9)

Output 08:

Warehouse Relation: R

no of possible Mappings -- 1

Mapping No:: 1

join of two source relations
And it is a partial projection on join

Equi join with join attributes --
Join Attribute(s)::

D
C

Source Name -- S1
Relation Name -- R2

Common Attributes::

E
D
C

Source Name -- S1
Relation Name -- R1

Common Attributes::

D
C
B

Case 09: projection or join

schemas::

source S1 true ORACLEOMEGA

relation R1

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)
attribute E char(9)

relation R2

attribute D char(9)
attribute E char(9)
attribute F char(9)
attribute G char(9)
attribute H char(9)

warehouse DW

relation R

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)
attribute E char(9)

Output 09:

Warehouse Relation: R

no of possible Mappings -- 2

Mapping No:: 1

a single source relation projection
And it is a complete projection of the source relation

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
E
D
C
B

Mapping No:: 2

join of two source relations
And it is a partial projection on join

Equi join with join attributes –

Join Attribute(s)::

E
D

Source Name -- S1
Relation Name -- R2

Common Attributes::

E
D

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
E
D
C
B

Case 10: union / intersection

schemas::

source S1 true ORACLEOMEGA

relation R1

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)

relation R2

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)

warehouse DW

relation R

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)

Output 10:

Warehouse Relation: R

no of possible Mappings -- 4

Mapping No:: 1

a single source relation projection
And it is a complete projection of the source relation

Source Name -- S1
Relation Name -- R2

Common Attributes::

A
D
C
B

Mapping No:: 2

a single source relation projection
And it is a complete projection of the source relation

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
D
C
B

Mapping No:: 3

join of two source relations
And it is a complete projection on join

Equi join with join attributes --
Join Attribute(s)::

A
D
C
B

Source Name -- S1
Relation Name -- R2

Common Attributes::

A
D

C
B

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
D
C
B

Mapping No:: 4

union or intersection of two source relations

Source Name -- S1
Relation Name -- R2

Common Attributes::

A
D
C
B

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
D
C
B

Case 11: synonyms

schemas::

source S1 true ORACLEOMEGA

relation R1

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)

relation R2

attribute E char(9)
attribute F char(9)
attribute G char(9)
attribute H char(9)

warehouse DW

relation R

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)
attribute F char(9)
attribute G char(9)
attribute H char(9)

synonyms::

S1.R1.D :: S1.R2.E

Output 11:

Warehouse Relation: R

no of possible Mappings -- 1

Mapping No:: 1

join of two source relations
And it is a complete projection on join

Cartesian join with no join attribute --
Source Name -- S1
Relation Name -- R2

Common Attributes::

H
G
F
E

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
D
C
B

Case 12: homonyms

schemas::

source S1 true ORACLEOMEGA

relation R1

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)

relation R2

attribute A char(9)

attribute B char(9)
attribute C char(9)
attribute D char(9)

warehouse DW

relation R

attribute A char(9)
attribute B char(9)
attribute C char(9)
attribute D char(9)

homonyms::

C :: S1.R1 ; S1.R2
D :: S1.R1 ; S1.R2

dwmapping::

DW.R.C :: S1.R1.C
DW.R.D :: S1.R1.D

Output 12:

Warehouse Relation: R

no of possible Mappings -- 2

Mapping No:: 1

a single source relation projection
And it is a complete projection of the source relation

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
D

C
B

Mapping No:: 2

join of two source relations
And it is a partial projection on join

Equi join with join attributes --
Join Attribute(s)::

A
B

Source Name -- S1
Relation Name -- R2

Common Attributes::

A
B

Source Name -- S1
Relation Name -- R1

Common Attributes::

A
D
C
B

REFERENCES

1. Madhavan, J., P.A. Bernstein, and E. Rahm, *Generic Schema Matching with Cupid*. 2001, Microsoft Corporation.

2. Rahm, E. and P.A. Bernstein, *On Matching Schemas Automatically*. 2001, MSR Tech. Report. p. 5-18.

3. Li, W.-S. and C. Clifton. *Semantic Integration in Heterogeneous Databases Using Neural Networks.* in *20th VLDB Conference*. 1994. Santiago, Chile.

4. Doan, A., P. Domingos, and A. Halevy. *Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach*. in *SIGMOD*. 2001.

5. Mitra, P., G. Wiederhold, and J. Jannink. *Semi-automatic Integration of Knowledge Sources*. in *FUSION*. 1999.

6. Milo, T. and S. Zohar. *Using Schema Matching to Simplify Heterogeneous Data Translation*. in *VLDB*. 1998.

7. Palopoli, L., G. Terracina, and D. Ursino. *The System DIKE: Towards the Semi-Automatic Synthesis of Cooperative Information Systems and Data Warehouses*. in *ADBIS-DASFAA*. 2000: Matfyzpress.

8. Bergamaschi, S., S. Castano, and M. Vincini, *Semantic Integration of Semistructured and Structured Data Sources.* SIGMOD, 1999. **28**(1).

9. Ursino, D., *Semi-automatic approaches and tools for the extraction and the exploitation of intensional knowledge from heterogeneous information sources*, in *Dipartimento di Elettronical Informatica e Sistemistical*. 1999, Universitá delgi Studi della Calabria. p. 15-34.

10. Doan, A., P. Domingos, and A. Levy, *Learning Source Descriptions for Data Integration*, University of Washington: Seattle, WA.

11. Clifton, C., E. Housman, and A. Rosenthal. *Experience with a Combined Approach to Attribute-Matching Across Heterogeneous Databases*. *Proc. 7th IFIP Conf. On DB Semantics*. 1997: Chapman & Hall.

12. Berlin, J. and A. Motro, *Database Schema Matching Using Machine Learning with Feature Selection*.

BIOGRAPHICAL INFORMATION

Karthik Jagannathan was born May 14, 1977 in Coimbatore, India. He received his Bachelor of Architecture degree from The School of Architecture and Planning, Anna University, Madras, India in June 1999. In the Fall of 1999, he started his graduate studies in Computer Science and Engineering at The University of Texas, Arlington. He received his Master of Science in Computer Science and Engineering from The University of Texas at Arlington, in December 2002. His research interests revolve around data warehousing and schema matching.