Automatic Design of Relational Databases

Solveig Torgersen
Ph.D Thesis

TR 89-1038
September 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

AUTOMATIC DESIGN OF RELATIONAL DATABASES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Solveig Torgersen

August 1989

AUTOMATIC DESIGN OF RELATIONAL DATABASES

Solveig Torgersen, Ph.D.

Cornell University 1989

Advances in relational database technology have made available relational database systems that support state of the art query languages and query processing algorithms. However, because database systems do not include adequate tools for database design, their use and accessibility is hampered by major difficulties that users experience in the process of designing a database. In this dissertation, we present methods and algorithms for automating the design of relational databases and describe a prototype design tool. Unlike researchers who have advocated the use of higher level data models, we focus on the pure relational model, because of its sound theoretical foundation and investigate issues in automatic design of relational databases.

We present a new graph representation for functional dependencies, which simplifies and enhances several design algorithms, such as algorithms for computing closures, keys, and projecting dependency sets. We define the basis $B(F)$, a compact representation of F^+, which is used to find multiple BCNF decompositions. The basis also provides a way to find the generator of the F-closed sets, an essential component in the computation of Armstrong relations, which are relations representing a set of functional dependencies.

We study the inference of multivalued dependencies from an acyclic relational scheme. These multivalued dependencies capture the relationship between the relations in the database. For the inference of functional dependencies within a relation, we optimize previously proposed algorithms.

Queries can be used to rate candidate schemes according to how queries perform against them. We present an algorithm for finding the exact query formulation for a particular design, given a scheme independent definition of the query. Multiple ways of accessing the data or no way of accessing the data consistently can be a result indicating that the current design is not valid.

Until now, there has been limited experience with feasibility and performance aspects of automatic relational design. We describe a prototype design tool and present a detailed performance study of the dependency inference algorithms implemented in this prototype.

Biographical Sketch

Solveig Torgersen was born in Pittsburgh, Pennsylvania, on December 15, 1957. She attended the University of Oslo in Norway, where she received the degree Cand.mag. in Mathematics and Informatics in December 1980. In 1983, she earned the Master of Science degree in Computer Science from Cornell University. She then joined Kvaerner Engineering AS in Norway for two years before she returned to Cornell University to continue her studies in Computer Science.

To my parents Eleonora and Bjarne
and my brother Dag Torger

Acknowledgements

I would like to thank Dina Bitton, my advisor, for her encouragement and advise. This dissertation would certainly not have been possible without her support. I thank David Gries and Frank Luk for serving on my committee. I would also like thank Randy Ames, Jeffrey C. Millman, Eliot Morrison, and Michael Laszlo for editing drafts of my dissertation.

I am grateful to my parents and brother for their support and understanding throughout the years of my studies. I also would like to thank Amr el Abbadi, Ken Birman, Rogerio Drummond, Janet Head, Jim Hook, Tommy Joseph, Rick Palmer, T.K. Srikanth, Anne Neirynck, Leena-Maija Reissell, and Aliza Wechsler for making my years at Cornell enjoyable. Thanks also go to my friends in Chicago; Lisbeth and Kenneth Andresen, Laila and Olav Bjarte Bergheim, Möyfrid and Ola Bjorland, Hedda Durnbaugh, Cyril Orji, and Arien Young.

Lastly, I wish to acknowledge the financial support I received from The Norwegian-American Association and The Norwegian Council for Science and The Humanities.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

During the two last decades, since Codd proposed the relational data model [Codd70], a substantial theory has developed for relational databases. This theory formalizes the concept of a well-structured relational database and provides a sound basis for query languages. The success of the relational database systems is largely due to the simplicity of the relational model combined with the power of relational languages. Relational database technology has recently advanced to the point that a broad range of highly tuned commercial relational database systems are available, ranging from systems on personal computers to high performance systems on mainframes and both parallel and distributed computers.

Unfortunately, since database systems do not include adequate tools for database design; hence, users experience difficulties in the process of designing a database. Designing a well-structured relational database is a complex and time-consuming task, which requires extensive knowledge of database design theory, as well as thorough understanding of the application for which the database is designed.

In this dissertation, we present methods and algorithms for automating the design of a relational database. We identify and address a number of fundamental problems that have hampered the development of automated tools for relational database design. We further describe solutions implemented in a prototype, *Design By Example*.

A relational database consists of a collection of *tables*, each having assigned a unique name and a number of columns. A column contains the values of an *attribute*, and a row represents a *tuple* of values that are related. Each table represents either an entity set or a relationship between entity sets. Designing a relational database essentially involves identifying the attributes that will form each table. For an entity-set table, the attributes describe the entity. For a relationship table, they identify the entities that are related. Attributes that are common to two tables establish an implicit relationship that can be made explicit by performing a *relational join*. A well-designed relational database is one in which the tables are structured so that the database is free from *data redundancies* and *update anomalies*.

The theory of relational database design [Ullm82, Maie83] has formalized the notion of a well-structured database by using the concept of *logical data dependencies*. A logical dependency is a mapping constraint between attributes, which can be *asserted* from the semantics of the database application. Data redundancies and update anomalies result when tables are not *normalized* (i.e., when tables are structured so that certain types of logical dependencies are allowed between attributes within a table or across multiple tables).

Although relational design theory is mathematically sound and well-conceived, it is not very much used in practice. Users of database systems, ranging from novice users of personal computers to professional programmers, do not usually have the mathematical sophistication required to understand and use relational theory. Furthermore, even the expert designer may have difficulties in specifying the input required by the design algorithms and in choosing one among the many candidate database schemes that these algorithms may produce. Current relational database systems do not provide tools or methodologies to assist a user in the complex process of designing a relational database. As a result, databases are often designed in an ad hoc manner, or in the case of complex database applications, database design is done manually by expert designers.

In this dissertation, we propose methods and algorithms for automating relational database design. In particular, we focus on algorithms and tools to help the database designer in the specification and manipulation of logical dependencies. Previous work on database design tools has usually been based on data models other than the relational one. It has been argued that the relational model lacks expressive power and that users experience great difficulties in specifying the input required by relational design algorithms, attributes, and logical dependencies. Furthermore, the conventional relational design algorithms are often inefficient and do not systematically generate multiple candidate schemes that are possible for a given set of logical dependencies. These reasons, combined with the complexity of relational design theory, have hampered the development of relational database design tools. Until now, there has been limited experience with feasibility and performance aspects of automatic relational design. Existing tools often limited and only have a graphical interface for the specification of entities and relationships and mapping a user-specified diagram to a partially normalized relation scheme.

In this dissertation, we contend that an expert relational design tool can automatically and transparently make available a number of important theoretical results, currently not available under data models other than the relational one. This tool should provide assistance in the specification of logical dependencies, which is one of the major difficulties in relational design. It should be able to tolerate poorly specified logical dependencies and assist the designer in elucidating and correcting the specifications. Finally, the tool should store and manage the design information in an efficient way. Since most relational design algorithms are inherently exponential, the tool should have heuristics and optimizations to speed up the interactive execution of the algorithms whenever possible.

The main contribution of this dissertation is a novel approach to the representation and manipulation of relational design information. We begin by taking a critical look at conventional relational design algorithms and propose new graph representations for functional and multivalued dependencies. These representations can be used internally by a design system to support efficient implementation of the design algorithms. They simplify some of the most commonly used design algorithms, such as computing closures, keys, and projections of dependency sets. Furthermore, they make the similarities and common components of the different design algorithms become more apparent. This can simplify the process of understanding the design algorithms and has also improved their time complexity.

Then, for the interactive component of the design tool, we present a flexible and user-friendly representation of logical dependencies in the form of example relations and example queries. By examining the example relations and queries, the designer can uncover anomalies in the data resulting from missing or incorrectly specified dependencies. Alternatively, the dependencies satisfied in an example table can be inferred automatically from the table. Inferring dependencies from example tables and generating small example relations from a given a set of dependencies, are both inherently exponential problems. We present a new method to find a small example relation, which is an exact representation of a set of dependencies. For the inference problem, we describe optimizations to existing algorithms and then present a feasibility and performance study of these inference algorithms.

Another contribution of this dissertation is a method for integrating the design of queries and the design of the database. The queries contain knowledge of how the database will be used. This information can be useful during the logical design phase since the logical design will affect the performance of the queries. Thus, we contend that the specification of the relational database queries should and can be viewed as a part of the logical design. Queries impose implicit constraints as to the natural grouping of attributes in the database scheme. We also investigate the possibility to infer dependencies that are satisfied among relations and not only within a specific relation. This inference can be useful when a database designer is forced to redesign an already existing database. The problem is equivalent to finding multivalued dependencies induced by a join dependency. We give a square algorithm to find a set of multivalued dependencies equivalent to an acyclic join dependency.

This dissertation is organized as follows. In Chapter 2, we describe the database design problem. In Chapter 3, we present concepts related to functional dependencies and give a survey of previous research on functional dependencies. Two new graph representations are introduced and a number of relational design algorithms (such as algorithms for finding closures, projections, and keys) are described based on these representations. Also, algorithms for finding a small example satisfying a given dependency set and for finding several decompositions in Boyce-Codd normal form are presented. In Chapter 4, we examine multivalued dependencies and how they can be induced from a join dependency or a relation scheme. Ways of finding join paths for a query over a specific scheme are presented in Chapter 5. In Chapter 6, we describe the implementation of a design tool and present a feasibility and performance study of the inference algorithms implemented in this tool. Chapter 7 is a summary and description of future work.

CHAPTER 2

Approaches to Database Design

A database management system can be divided into two levels of abstraction at which data in the database system may be viewed, the physical and the logical. The physical level deals with the actual storage and representation of information in the database. The logical level provides the model from which the user views this information. During the process of designing a database, the logical and physical designs are chosen. The chosen design will affect the performance and ease of use of the database system. Research in database design has concluded that the logical and physical design problems depend on the semantics of the database application and on the queries to this database [Ullm82, Maie83a].

In this chapter, we describe different approaches to database design, in particular the logical and conceptual design problem. In Section 2.1, we define the design problem, in Section 2.2, we describe the Entity-Relationship model, and in Section 2.3, we describe the Iris model. We conclude the chapter with an introduction to our approach to database design and argue for its advantages compared to the higher-level models.

2.1. The Database Design Problem

A database system should provide an efficient way to retrieve and store information. The efficiency of a particular database does not solely depend on its management system, but also on the logical organization of the data. This leads us to the major concern of *database design*: to organize the data so that the database system can efficiently satisfy the users processing requirements, to operate with high performance, to be space efficient and data consist, and to eliminate insertion and deletion anomalies. Organizing the data to satisfy these requirements begins with a thorough knowledge of the database application and its semantics. To simplify the difficult, complex, and time-consuming task of designing databases, the design process is usually divided into four major phases [Yao85]: (1) *requirement analysis and specification*, (2) *conceptual design*, (3) *logical design*, and (4) *physical design*. In the rest of this section, we will describe these phases.

2.1.1. The Requirement Analysis and Specification

During the requirement analysis and specification phase, detailed information about the application is collected through interviewing the potential users and the application specialist. The specifications are then usually written in essay form. It is important that the specifications are as complete, consistent, and understandable as possible, since the next phases in the design process depend on them. Failure to specify the goals and the requirements often result in designs that do not satisfy their intended purpose. Because of the number of people and the amount of data that can be involved, design tools are useful in the specification phase. The most common tool is the *Data Dictionary*, a tool in which definition, structure, and use of data can be stored and retrieved. The requirement analysis and specification phase is a research topic by itself and we will not be investigate it any further but refer to [Yao85] for further information.

2.1.2. The Conceptual Design

During the conceptual design phase, the information from the specification phase is modeled in a unified and systematic way. The conceptual design process usually includes mapping the specifications into a *semantic model*. The semantic models are used to model the application rather than the database. They provide modeling constructs to express the meaning of the data in a way users will easily understand. The growing need to design large and complex database systems has stimulated the development of semantic models that use higher level concepts and allow the specification of complex relationships in the data. These powerful semantic models can more precisely illustrate the application than the logical models can, but finding a good logical design for the database remains difficult. There are many semantic models; however, we shall study two of them in Sections 2.2 and 2.3.

2.1.3. The Logical Design

During the logical design phase, the logical structures, defined by the data model of the database, are formulated from the conceptual design. The three most common data models are the hierarchical, network, and relational models [Ullm82]. Recently, the object-oriented model was introduced. The logical model allows the user to specify operations on the data in the database without requiring detailed understanding of the physical representation of the data. The organization of data into tables or records depends on the logical data model of the database management system and directly affects how the data is logically accessed. The logical design also determines whether the specifications are satisfied, whether redundancies are eliminated, and whether the tables are normalized.

2.1.4. The Physical Design

In the final stage, the physical design phase, the appropriate storage structure for a given logical design is chosen. The database management system usually supports a variety of storage structures or access methods and allows various parts of the database to be stored differently. For instance, in a relational database, one relation may be stored as a B-tree and another relation may be hashed with respect to its key field. The problem of selecting appropriate storage structures for the relations in the database is called the index selection problem. Different index selections will render different performance characteristics. To find an appropriate physical design, the designer must understand how the database will be used and know the relative frequency of the different queries. It is practically impossible to find an optimal physical design for a database with many queries. Comer [Come78] proved that a simplified version of the index selection problem is NP-complete. Most database management systems allow changes to the physical structures of the database in use.

All phases of database design require involvement from application specialists as well as database design experts. The decisions done in one phase can affect the final design, and some of the phases may need to be modified in order to achieve a better end-product. The semantic models were introduced to simplify the design task and to provide an interface between application specialists and database design experts. These models vary depending on what aspect of conceptual design they emphasize. The earliest models, such as the Entity-Relationship model [Chen76] and the Functional Data model [Ship81], are mainly concerned with the static meaning of an application. More recent models, such as TAXIS [Mylo83] and Galileo [Alba85], focus on features that can model dynamic behavior of data. Both TAXIS and Galileo are languages, with philosophical similarities to Pascal, which include flow of control facilities and arithmetic capabilities. These so-called *database programming languages* can act as query and application-specification languages, thus providing the flexibility of general-purpose programming constructs for specification of database transactions.

Below, we present two models that have been used for the conceptual design of databases. The first is the Entity-Relationship model (ER-model); the is the Iris data model [Derr85]. We refer to others [Hull87, Peck88] for a more extensive discussion and survey on semantic data models.

2.2. The Entity-Relationship Model

The ER-model initially introduced by Chen [Chen76] was one of the early conceptual models. This model attempts to combine the best features from the network model and from the relational model. In order to capture more semantic meaning, a variety of extensions have been suggested for this model [Ling85, Teor86]. In its most recent variant, the ER-model is called the extended entity-relationship model (EER-model) [Teor86]. The ER-model is the most popular conceptual model and a number of design systems use it. Some of these systems are the ER-Designer by Chen & Associates, MAST-ER by InfoDyne, and SunUnify by Sun Microsystems.

As the name suggests, the ER-model's main constructs are *entities* and *relationships*. These constructs are depicted in an ER-model as boxes and diamonds. An entity is represented by a rectangular box and a relationship by a diamond box. Entities, and sometimes the relationships, have *attributes* that describe their properties. Each attribute is associated with a value from a domain, usually a set of integers, real numbers, or character strings. In the ER-model, an attribute is represented by an oval.

An entity is an object that can be uniquely identified in the application. An example of an entity for a university database is a specific teacher or student. Entities that satisfy a set of predefined common properties are grouped into *entity types*.

Example 2.1: Teachers and students can be grouped into one entity type Person, but if the application distinguishes teachers from students and associates different properties with them, we would create two entity types, Teacher and Student. □

Relationships are associations among one or more entities. These can be many-to-many, one-to-many, and one-to-one. The original ER-model allowed only binary relationships, involving only two entities. The EER-model allows *n*-ary relationships where several entities can be involved.

Example 2.2: In a university environment, teachers teach courses. The ER-model of the binary relationship Taught-By, between the entities Teacher and Course, is shown in Figure 2.1. □

The number of entities in a relationship is called the *degree* of the relationship. The *connectivity* of a relationship specifies the mapping of the entities occurring in the relationship. Let us say that for a relationship over entities $E_1, E_2, ..., E_p$ has connectivity one for entity E_i, this means that only one entity from E_i is related to the other entities. The actual number associated with the connectivity is called the *cardinality* of the connectivity. The occurrence of an entity in a relationship can be *optional* or *mandatory*, indicating whether null values are allowed or not.

Example 2.3: Consider relationship Taught-By given in Example 2.2 and depicted in Figure 2.1. This relationship has degree two and the cardinality of the connectivity for Teacher is "m" and for Course is "n". □

There are two types of attributes, *identifying* and *descriptive* . The identifying attributes are essential to identify a particular entity; such attributes are said to be a *key* or to belong to a key, since they are used to identify an entity. The identifying attributes are usually represented by underlining the attribute name. Attributes that cannot uniquely identify or be used to identify entities are considered descriptive attributes.

There are two types of entities, *strong* and *weak*. A strong entity can be identified uniquely by some of its attributes, while a weak entity is identified through

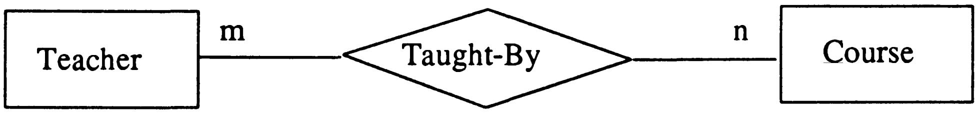


Figure 2.1. Relationship Taught-By

its relationship with other entities. We can say that there is an existence relationship between the weak entity and the strong entity on which it depends. A strong entity has an identifying key, while a weak entity does not. In an ER-model the weak entity is represented as a double rectangular box.

Example 2.4: Consider entities City and Street. A street, say "State Street," can be in many cities, and a particular "State Street" can be uniquely identified only through a given city. We say that the entity Street is weak since its unique identification depends on an entity from entity type City. □

To allow an entity to be a specialization or a generalization of another entity the ER-model has been extended with two additional object types, specialization hierarchies and generalization hierarchies [Nava83, Elma85]. These concepts are defined as follows:

Definition 2.1: *Specialization Hierarchy* [Teor86]. An entity E_1 is a specialization of another entity E_2, if every occurrence of E_1 is also an occurrence of E_2.

Definition 2.2: *Generalization Hierarchy* [Teor86]. An entity E is a generalization of the entities $E_1, E_2, \ldots, E_n$ if each occurrence of E is also an occurrence of one and only one of the entities $E_1, E_2, \ldots, E_n$.

Specialization makes it possible to single out a group of entities from an entity type that satisfies some additional properties. All entities that are of entity type E_1, a specialization of entity type E_2, are also contained in E_2. The generalization, on the other hand, makes it possible to view parts of information from a particular entity type.

Example 2.5: Consider the entities Student, Teacher, and Person. We can say that Student and Teacher are specializations of Person since both of these specializations are subsets of Person and neither of them contains all entities in Person. Let the Person entity contain all the information of persons associated with the university, including students and teachers. A Directory entity containing only the person name and phone number is a generalization of the Person entity. □

There have been several approaches to transform an ER-model into the relational model. One approach [Teor86] is to use empirical design rules while designing the ER-model. If the rules are carefully followed, the resulting ER-model can be translated into the relational model. The other approach is to normalize the ER-model [Ling86] before the transformation is done without restricting the database designer to model the database in a special way. The objective of both approaches is to assure that the ER-model can be translated into relations that are normalized, usually in 3NF or 5NF (see chapters 3 and 4, respectively). The semantics of the ER-model will then be expressed in terms of functional dependencies (see Chapter 3). The actual transformation of the ER-model to relations consists of the following steps:

1. Transform every entity into one relation with the same attributes as the entity.
2. Transform every relationship into a relation with attributes containing the keys of the entities involved in the relationship and the attributes of the relationship.
3. Derive the functional dependencies from the relationships in the ER-model (as explained below).
4. Remove relations that are subsets of other relations.

In the case where the ER-model is not normalized, the designer must perform additional checks on the design to see that the relation schemes are normalized. If the relations are not normalized, then conventional algorithms for normalization are used.

Functional dependencies, representing one-to-one and many-to-one relationships, can be derived from the relationships of an ER-model. A relationship that has an entity whose cardinality is one will produce a functional dependency. A many-to-one relationship from entity A to entity B will produce the functional dependency A→B.

Consider the relationship Taught-By in Figure 2.1. This is a many-to-many relationship and no functional dependencies can be derived. If we change the cardinality of the entities in the relationships, the following three cases may occur:

Course with Cardinality One
: This means that a teacher can teach one course but many teachers can teach the same course, giving the functional dependency Teacher→Course.

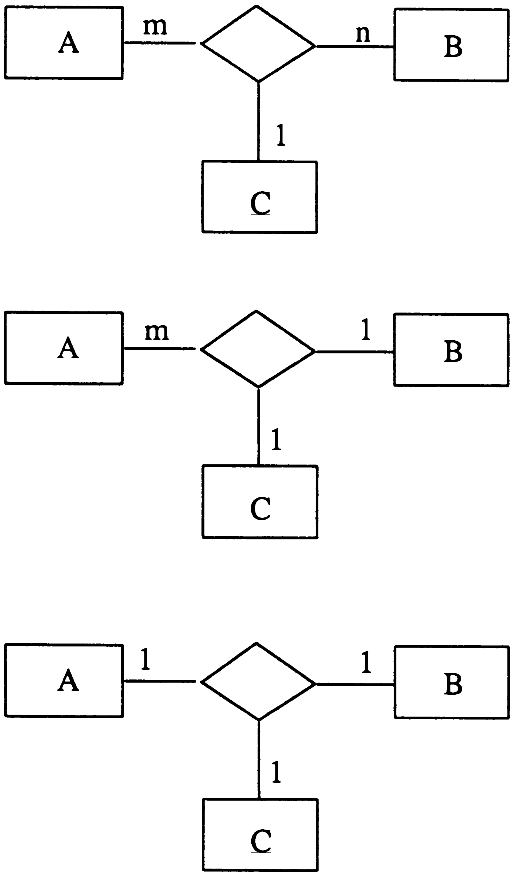
Teacher with Cardinality One
: This means that a course can be taught by one teacher but a teacher can teach many courses, giving the functional dependency Course→Teacher.

Both Teacher and Course have Cardinality One
: This means that a teacher can teach one course and a course can be taught by only one teacher, giving the functional dependencies Teacher→Course and Course→Teacher.

It is more difficult to derive functional dependencies from relationships with degree 3 or higher. In fact, different researchers propose different rules for deriving functional dependencies from ternary relationships. In Figure 2.2, we give the functional dependencies as Ling [Ling85] and Teorey et al. [Teor86] derive them.

Figure 2.3 shows the ER-model for the university database. ER-models with many entities and relationships tend to be complicated, but most of the entity-relationship design tools allow view and manipulation of parts of the diagram. Transforming this ER-model into the relational model according to the rules above will result in a scheme with the nine relations, which are:

(Room-No, Building)
(T-Name, SSN)
(S-Name, SSN)



Ling	Teorey et al.
AB→C	AB→C
A→BC	AB→C AC→B
A→BC B→AC C→AB	AB→C AC→B BC→A

Figure 2.2. FDs Derived from Ternary Relationship

(C-Name, C-No)
(Hour, Day, Room-No, T-Name)
(Hour, Day, Room-No, S-Name)
(C-Name, T-Name)
(S-Name, C-Name, Grade)
(Room-No, Hour, Day, C-Name)

We have 10 entities and relations, but the relation initially created by the Time entity was removed since both its attributes are contained in relations created due to the relationships containing Time. The functional dependencies that can be derived from this ER-model are:

Taught-By: C-Name→T-Name

Room-Schedule: Hour, Day, Room→C-Name
Lecture-Room: S-Name, Hour, Day→Room
Lectures-In: T-Name, Hour, Day→Room

Additional dependencies hold within each of the entities Student, Teacher, and Course since these entities have the unique identifiers S-Name, T-Name, and C-Name respectively. The relation resulting from the relationship Course-Grade will satisfy the functional dependency S-Name, C-Name→Grade.

The ER-model gives a good way of describing a conceptual design. But, in contrast to the newer conceptual models one can only describe static information. The ER-model is also relatively easy to transform into the relational or network model. Its simplicity has contributed to its popularity. Despite the ease of using the ER-model,

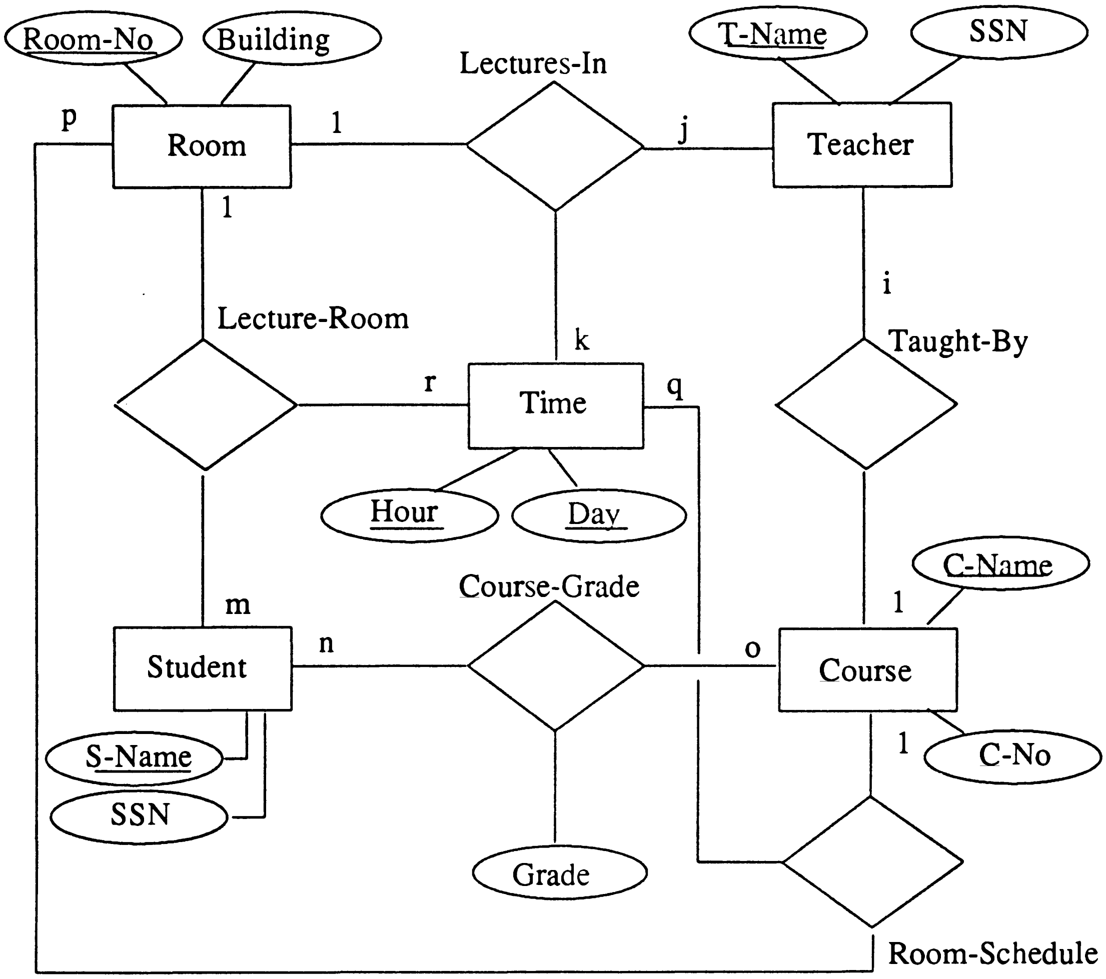


Figure 2.3. ER-Model for University Database

the design task still requires a considerable understanding of the application and of the database design process. The specification of a correct set of dependencies remains and the ER-model provides no help or guidance in the process.

2.3. The Iris Data Model

The Iris data model [Derr85] introduced by Derret, Kent, and Lyngbaek, recently developed and implemented at Hewlett-Packard Laboratories, is a semantic data model that is designed to support high-level abstractions as well as behavioral abstractions. The model has evolved from the Integrated Data Model (IDM) developed at Hewlett-Packard Laboratories [Beec83], with features from DAPLEX [Ship81] and TAXIS [Mylo83]. DAPLEX supports the functional model and query specification using the basic elements of first-order predicate calculus. The TAXIS language is, as mentioned earlier, a database programming and specification language.

An Iris diagram is defined as a directed graph extended with a set of constraints. The graph consists of *objects*, *types*, and *functions*. An object represents a specific instance of an entity or a concept in an application. There are two kinds of objects, *literal* and *non-literal*. A literal object is represented by an exact value such as an integer, a real, or a string. A non-literal object is an object that does not have a direct value representation, such as a person. Sets of objects are grouped together into types, allowing different types to contain the same objects. Types are literal or non-literal, depending on the kind of objects they represent. In an Iris diagram, literal types are represented by boxes and non-literal by circles.

Example 2.6: Consider the types Student and Grade. We say that "John Doe" is a non-literal object from the non-literal type Student. "A" is a literal object from the literal type "Grade." □

The non-literal types can be organized to support generalization and specialization by using subtype and supertype relationships. All the instances of a subtype are also instances of a supertype. In an Iris diagram this is represented as a directed edge from the subtype to the supertype.

Information about objects is modeled using predicate functions. For example, the fact that a person has a name is represented as a predicate connecting the person object and the name object. This approach is different from the ER-model where objects have attributes. The attribute concept is modeled in Iris by using functions whose values are derived from the predicates. For instance, a predicate *Person-Age* that connects persons and their ages is defined as follows:

Create function *Person-Age*(**Person** *p*, **Integer** *a*);

where Person-Age(p,a) is true iff person p has age a. Functions may have several argument types and result in many types. A function is graphically represented by a directed labeled edge starting in *cross-product* vertices representing the arguments and ending in cross-product vertices representing the range. The cross-product vertex is graphically represented as a circle surrounding a cross.

Example 2.7: Consider types Student, Grade, and Course. The Iris diagram representing function Course-Grade: Student × Course → Grade is given in Figure 2.4 a. □

Sometimes an edge between a type vertex and a cross-product vertex is labeled with a *role* name, different from the type that the edge is coming from. This is useful in the case argument or range types in the function coming from the same type, but play different roles or have different meanings in the function. This will also reduce the number of types defined in the Iris diagram. Edges between type and cross-product vertices that are not labeled are considered to have the role of the type vertex.

Example 2.8: Consider types Student, Course, and Letters. Instead of having a type Grade we can use type Letters instead. Function Course-Grade of Example 2.7 is illustrated in Figure 2.4 b, but then the function is written as Course-Grade: Student × Course → Grade/Letter to indicate that type Letter plays the role as Grade in the function. □

It is also possible to specify *object participation constraints* that accompany the function. The object participation constraints are meant to characterize the function as a multivalued, single valued, partial, or total function. The participation constraint of a function f is a set of expressions $X[l,u]$ where X is the set of labels of role edges connected to the cross-product vertices of function f. Parameter l indicates whether the function is total or partial; in other word, whether all or only some objects are included in a relation consisting of the types (attributes) used in the function.

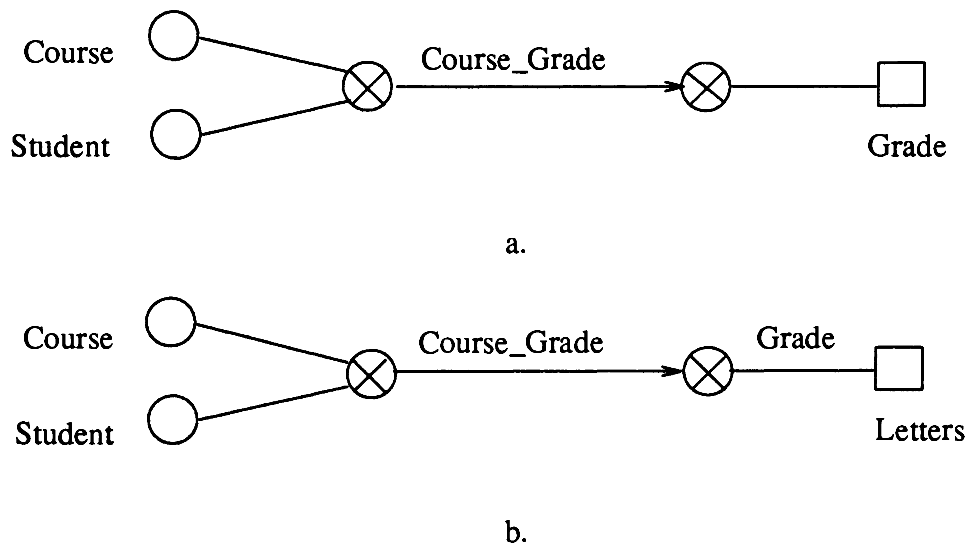


Figure 2.4. Function Course-Grade

Parameter u indicates whether the function is single or multivalued; more specifically, it gives an upper bound on the number of times an object or a set of objects occurs in a relation consisting of the types (attributes) in the function.

Example 2.9: Consider the function social security number:

SSN: Person→String

and the following participation constraints:

SSN: Person[1,1]

SSN: String [0,1]

The first constraint means that a person has exactly one social security number, no more and no less. The second constraint means that the social security is unique but not all social security numbers are in the relation SSN (Person,String). □

Example 2.10: Consider again the function

Course-Grade: Student × Course → Grade/Letter.

We can define the following participation constraints:

Course-Grade: Student, Course [0,1]

Course-Grade: Student [0,5]

Course-Grade: Course [1,100]

They tell us that a student has at most one grade in a course, that a student attends at most five courses, and that a course is attended by at most 100 students. □

Figure 2.5 shows an Iris diagram and functions for a university database. We see that even for simple designs, the diagram can be rather complex. The Iris design tool allows the user to view one part of the diagram at a time.

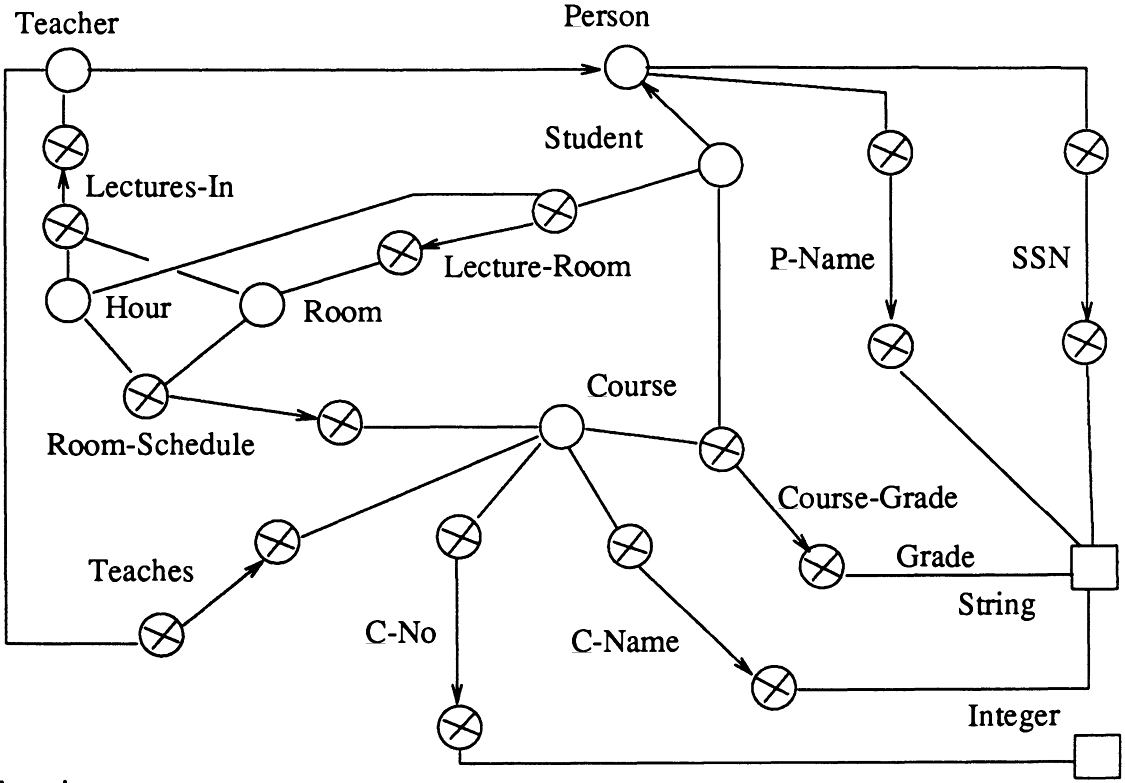
The Iris diagram is usually stored as a relation scheme where a relation corresponds to the objects of a function. In addition, it is possible for the user to specify objects that should be clustered together to form a relation. An Iris diagram can thus be translated into a relational scheme, but it is not always possible to produce relations that will be in Boyce-Codd normal form (see Section 3.1). This is mainly due to functions with multivalued participation constraints. Allowing a rich set of integrity constraints has produced problems in conforming to the relational model. Another problem with the model is that rules to derive functions from other functions have not been as successful as for the functional dependencies and multivalued dependencies (see Sections 3.1 and 4.1). Due to the inability to find sound mathematical rules to translate Iris diagrams into normalized relations, there is no guarantee that the Iris model will produce a good relational database.

The Iris diagram supports specification of queries, which are expressed in terms of functions and objects. The specification uses an extended form of SQL in the form

select(T_1 x_1 , ..., T_n x_n) where *predicate*

Variables x_1 , ..., x_n are called *result variables* of types T_1 , ..., T_n, respectively.

Example 2.11: Using the Iris diagram for the University Database, the query that finds all the students that received a grade A in any specific course is specified as



Functions:

Course-Grade: Student×Course→Grade/String	
Course-Grade: Student, Course[0,1]	C-Name: Course→String
Room-Schedule: Hour×Room→Course	C-Name: Course[1,1]
Room-Schedule: Hour,Room[0,1]	C-Name: String[0,1]
Lecture-Room: Student×Hour→Room	C-No: Course→String
Lecture-Room: Student, Hour[0,1]	C-No: Course[1,1]
Lectures-In: Hour×Room→Teacher	C-No: String[0,1]
Lectures-In: Hour,Room[0,1]	SSN: Person→String
P-Name: Person →String	SSN: Person[1,1]
P-Name: Person[1,∞]	SSN: String[0,1]

Figure 2.5. Iris Diagram for University Database

```
create function A-Student(Course c) as
    select s for each Student s
        where Course-Grade(s,c)='A'; □
```

Mapping the Iris queries to select-project-queries has been found difficult, especially if the where clause contains several nested functions. For further detail on Iris we refer the reader to [Lyng87].

2.4. Conclusion

Many database designers believe that the relational model is not powerful enough to be used as a conceptual model, since some applications cannot be mapped into relational tables in a straightforward fashion. Approaches based on data models of higher level unfortunately lack the sound mathematical foundation of the relational model. There has not been sufficient research on automating relational database design, and there is little experience with developing expert design tools for relational database design. Thus, further investigation using a pure relational model during the conceptual and logical design phase is of interest.

A major argument against the relational model and in favor of higher level models has been the difficulty that a database designer may experiences in specifying the input required by the relational design algorithms (the set of functional and multivalued dependencies). In the next two chapters we address this problem and propose a solution based on examples and automatically inferring dependencies from the sample data. Then, in Chapter 6, we will describe a prototype that implements this solution.

CHAPTER 3

Representations of Functional Dependencies

The design theory of relational databases has formalized through sound theoretical concepts the notion of a well-structured design. A number of algorithms have been provided to achieve good designs. The main concern of these algorithms is the manipulation of dependencies that describe the relationships between attributes in the application database. The normal forms that define a well-structured scheme are defined with respect to these dependencies and can be found by structuring the dependencies. The most commonly used dependency in relational design is the *functional dependency*, introduced by Codd [Codd72]. A functional dependency describes a one-to-one and many-to-one relationship between attribute sets.

Despite the simplicity of the concept of functional dependencies, applying them to relational design has often caused problems for database designers. Defining them for a particular application is often difficult, and manipulating them during the logical design process requires a fairly complex mathematical treatment. Unfortunately, the literature on logical design theory does not provide a unified treatment of functional dependencies in the design algorithms. This has resulted in a widening gap between theory and practice, with practitioners continuing to use empirical design rules rather than rigorous algorithms.

In this chapter we present two new graph representations for functional dependencies. The graph representations are extensions of directed graphs. The first is a lattice-like representation. We show that the functional dependencies can be ordered in a semilattice. This representation leads to the definition of a *basis* for functional dependencies that will form a foundation for several of the logical design algorithms presented in this chapter. We call this representation the "Semilattice Representation."

The second representation is a more direct way of viewing a particular functional dependency set. The representation is a directed graph with two types of vertices, those representing functional dependencies and those representing attributes. A directed edge from an attribute vertex to a functional dependency vertex indicates that the attribute is in the left-hand side of the functional dependency. Similarly an edge from a functional dependency vertex to an attribute vertex indicates that the attribute is in the right-hand side of the functional dependency. We call this representation the "Attribute-Fd" graph.

The attribute-fd graph is used to explore better algorithms and heuristics in logical design, such as algorithms for finding covers of functional dependencies, projecting dependencies onto a set of attributes, and finding keys. The graph theoretic approach provides a unified way of solving the above mentioned problems and underlines the similarities of the methods used to solve these problems.

Several of the algorithms that manipulate functional dependencies are inherently exponential in time; projecting dependencies and finding keys are examples. Despite this, we show that for a large class of functional dependency sets these algorithms have polynomial behavior.

In Section 3.1, we present useful concepts and terminology related to functional dependencies. Section 3.2 contains a survey of previous research on functional dependencies. In Section 3.3, we describe the semilattice representation of a functional dependency set and in Section 3.4, we describe the attribute-fd graph for functional dependencies. In Sections 3.5 through 3.10 we present efficient design algorithms using these two graph representations.

3.1. Concepts and Definitions

The functional dependency $X \rightarrow Y$ describes a one-to-one or a many-to-one relationship between attribute set X and Y. This means that each X-value is associated with at most one Y-values. Formally, a functional dependency is defined as follows:

Definition 3.1: *Functional Dependency.* We say that a relation **r** satisfies the functional dependency $X \rightarrow Y$, where X and Y are subsets of the set of attributes, if whenever **r** has two tuples that agree on all attributes of X, then these tuples also agree on attributes of Y. Thus, relation **r** satisfies the functional dependency $X \rightarrow Y$ iff for every tuple t_1 and t_2 where $t_1[X]=t_2[X]$, $t_1[Y]=t_2[Y]$.

Example 3.1: Consider the university database relation

R={Teacher, Course, Hour, Room, Student, Grade}.

A course is taught by one teacher and a student receives one grade in a course. This can be expressed by the functional dependencies

{Course}→{Teacher} and

{Course, Student}→{Grade}. □

Functional dependencies can logically imply other dependencies. We say that dependency set F logically implies $X \rightarrow Y$, written $F \models X \rightarrow Y$, if a relation that satisfies F also satisfies $X \rightarrow Y$. The functional dependency set that contains all the dependencies logically implied from F is called the *closure* of F, denoted by F^+. To find F^+ we can use the Armstrong inference axioms for functional dependencies [Arms74], given below. Note that the set R contains all the attributes of the relational database and that the notation XY means $X \cup Y$.

Reflexivity.
If $Y \subseteq X$, then $X \rightarrow Y$.
Union.
If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.
Transitivity.
If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.
Pseudotransitivity.
If $X \rightarrow Y$ and $YW \rightarrow Z$, then $XW \rightarrow Z$.
Decomposition.
If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$.
Augmentation.
If $Z \subseteq R$ and $X \rightarrow Y$, then $XZ \rightarrow YZ$.

The reflexivity, union, and transitivity rules have been proved to be complete, and the remaining rules are derivable from these three axioms [Ullm82].

An *Armstrong relation* [Arms74] is a special relation instance that can show both the nonexistence and the existence of dependencies for a relation. In other words , an Armstrong relation satisfies all dependencies inferred from F and violates all other dependencies [Silv81, Mann85, Bitt85]. Armstrong relations can be regarded as a representation of the closure of a dependency set.

It is useful to know which attributes are functionally determined by a particular attribute set X with respect to a set of functional dependencies F. The set of all attributes determined by X is called the *closure* of X, denoted by X^+. The closure is found by augmenting the attribute set using the transitivity rule, until no dependency can be applied. This can be done in $O(|F|)$ time , where $|F|$ is the number of attributes occurring in the dependency set. A set of attributes X is *F-closed* with respect to the functional dependency set F if all attributes A that X determines are members of X. This means that the closure of an F-closed set X is X. A set of attributes whose closure contains all the attributes of the relation is called a *key*.

Example 3.2: Considering the Example 3.1 with dependency set F={{Course}→{Teacher}, {Course, Student}→{Grade}}, we find that the closure of {Course, Student} is {Course, Student, Grade, Teacher}. The attribute set {Teacher, Hour, Room, Student, Grade} is a key for the relation. The sets {Teacher}, {Student}, and {Grade} are all F-closed but {Course} is not. □

Much research has focused on minimizing the size of the dependency set without changing the semantic meaning (F^+) of the set. Two functional dependency sets F_1 and F_2 are said to be *equivalent* if they have the same semantic meaning (i.e., $F_1^+ = F_2^+$). F_1 is defined to be a *cover* of F_2, if F_1 and F_2 are equivalent. Below we define some commonly used covers for functional dependencies.

Definition 3.2: *Nonredundant Set of Functional Dependencies.* A functional dependency set F is *nonredundant* if no proper subset of F is equivalent to it.

Definition 3.3: *Minimum Cover* [Maie80]. A set of functional dependencies F is *minimum* if there is no smaller set equivalent to it.

Definition 3.4: *Left Reduced Cover*. A set of functional dependencies is *left reduced* if for every functional dependency $X \rightarrow Y$ in F, there is no $X' \subset X$ with $X' \rightarrow Y \in F^+$. The attributes that can be removed are called *extraneous*.

Definition 3.5: *Right Reduced Cover.* A set of functional dependencies is *right reduced* if for every functional dependency $X \rightarrow Y$ in F, there is no proper subset $A \in Y$ such that $X \rightarrow A \in (F - \{X \rightarrow Y\} \cup \{X \rightarrow Y - A\})^+$. The attributes that can be removed are called *extraneous*.

Definition 3.6: *Canonical Cover* [Pare77]. A set of functional dependencies F is *canonical* if F is nonredundant and for every $X \rightarrow Y \in F$

1. $X \rightarrow Y$ is left reduced, and
2. Y is a single attribute.

Definition 3.7: *LR-minimum Cover* [Maie80]. A set of functional dependencies F is *LR-minimum* if the set is a minimum cover and it is both left and right reduced.

A nonredundant cover equivalent to a functional dependency set F can be found in $O(\|F\| \cdot |F|)$ time, where $\|F\|$ is the number of dependencies in the set and $|F|$ is the number of attributes occurring in the dependency set. Each dependency $X \rightarrow Y$ is checked whether $X \rightarrow Y \in (F - \{X \rightarrow Y\})^+$; if so, then $X \rightarrow Y$ is redundant in F. This takes $O(|F|)$ time for each dependency.

To find a canonical cover takes $O(|F|^2)$ time. It takes $O(|F|)$ time to transform F into the form with single attributes on the right-hand side, resulting in $O(|F|)$ dependencies. Thus, to make the set nonredundant takes $O(|F|^2)$ time. To remove the extraneous attributes from the left-hand side, that is to test for all $X \rightarrow Y \in F$ whether $(X-A) \rightarrow Y \in F^+$ and $A \in X$, takes $O(|F|^2)$ time. Thus it takes $O(|F|^2)$ time to find a canonical cover.

A minimum dependency set can be found in $O(\|F\| \cdot |F|)$ time. It takes $O(\|F\| \cdot |F|)$ time to compute the closure of all the left-hand sides. While computing the closure for a left-hand side X, one marks an $\|F\| \times \|F\|$ matrix, whose rows and columns are indexed by dependencies in F; a marked index (i,j) indicates that while finding the closure of left-hand side X_i, dependency $X_j \rightarrow Y_j$ was applied. Thus, after finding the closure of X_i one can find all the dependencies $X_j \rightarrow Y_j$ such that both $X_i \rightarrow X_j$ and $X_j \rightarrow X_i$ are in F^+. Given this matrix, it takes $O(\|F\|^2)$ time to find all dependency sets F_i such that $X_i \in X_j^+$ and $X_j \in X_i^+$. If X_i determines $X_j \in F_i$, with respect to $F - F_i$, the dependency set F can be minimized by removing dependency $X_i \rightarrow Y_i$ and adding Y_i to the right-hand side of dependency $X_j \rightarrow Y_j$. This minimization takes $O(\|F\| \cdot |F|)$ time. For details see [Maie80, Maie83a].

LR-minimum covers can be found in $O(|F|^2)$ time given the above results. It takes $O(||F||\cdot|F|)$ time to minimize F and $O(|F|^2)$ time to remove extraneous attributes from the left- and the right-hand sides of the functional dependencies.

As a matter of reference, we define the following normal forms:

Definition 3.8: *First Normal Form* (1NF). A relation scheme F is in first normal form if all attributes in the relation are atomic, that is attributes do not have composite values.

Definition 3.9: *Second Normal Form* (2NF). A relation scheme F is in second normal form if for all functional dependencies $X \rightarrow A$ that hold over R, A is either in a key or no subset of any key can determine A.

Definition 3.10: *Third Normal Form* (3NF). A relation scheme F is in third normal form if for all functional dependencies $X \rightarrow A$ that hold over R, X either contains a key for R or A is in a key.

Definition 3.11: *Boyce-Codd Normal Form* (BCNF). A relation scheme F is in Boyce-Codd normal form if for all functional dependencies $X \rightarrow A$ that hold over R, X contains a key for R.

Example 3.3: Consider the relation R={A,B,C,D} and the functional dependencies F={AB→CD, C→D, D→B}. This dependency set has keys AB and AD, since they are the minimal sets of attributes that determine R. The relation is in 2NF because all attributes except C are in a key and C can not be determined by any subset of the keys. The relation is not in 3NF or BCNF. If the second dependency is removed the relation would be in 3NF, and if only the first dependency is in the dependency set the relation would be in BCNF. □

3.2. Survey of Previous Research

The relational model and the first normal form were introduced by Codd in 1970 [Codd70]. To formalize the notion of well-formed relations, Codd defined functional dependencies [Codd72] as relationships between the attributes of a relation. Codd then proposed the second and third normal forms to define well-formed relations in terms of properties of functional dependencies.

Delobel and Casey [Delo73] described a set of inference rules for functional dependencies, and Armstrong [Arms74] refined these properties and showed them to be complete. These rules are commonly known as Armstrong axioms for functional dependencies. Every problem dealing with functional dependencies requires a manipulation of functional dependencies according to these rules.

Bernstein [Bern76] gave the classic third normal form decomposition algorithm with time complexity $O(|F|^2)$. However, this algorithm only reduced the functional dependencies by removing redundant dependencies. Paredaens [Pare77] introduced the definition of canonical covers for functional dependencies. Lewis, Sekino, and Ting [Lewi77] formulated a representation for all nonredundant covers of a set of functional dependencies.

Lucchesi and Osborn [Lucc78] presented an algorithm for finding all the keys of a given set of functional dependencies. They also showed that deciding whether an attribute is a member of a key is NP-complete.

Beeri and Bernstein [Beer79] gave both a polynomial and a linear time algorithm to find the closure of a set of attributes. They also described algorithms for removing extraneous attributes from the left-hand side of a functional dependency and for testing whether a dependency is redundant. Maier [Maie80] extended Beeri's and Bernstein's work to minimize the cover by removing a dependency through adding attributes to the right-hand side of another dependency and by removing extraneous attributes from the right-hand side of other dependencies. Maier also showed that finding the optimal cover is NP-complete. Jou [Jou80] and Beeri and Honeyman [Beer81b], found that to test whether a given set G of functional dependencies is a cover for a set of functional dependencies F is co-NP-complete.

Fischer, Jou, and Tsou [Fisc83] showed that finding the projection of a functional dependency set onto a relation is exponential in time and space, since covers for the projected dependency set can be exponential in size of the original dependency set.

Beeri et al. [Beer84b] defined the concept of F-closed set. These sets can be used to construct an Armstrong relation. Mannila and Räihä [Mann85] defined the *generator set* for all the F-closed sets of F and showed that this set could be used to find Armstrong relations more efficiently.

Recently, Gottlob [Gott87] gave an algorithm to find the projection of functional dependency sets onto a relation. The algorithm finds a cover for the projection of the dependencies without finding the closure of the dependency set F. Diederich and Milton [Died88] gave a heuristic algorithm for finding canonical covers that reduces the number of closures computed. This approach also reduces the number of closures computed while finding a 3NF decomposition.

Graph representation of functional dependencies have been previously used, but not extensively. Codd [Codd72] used a graph to visualize the functional dependencies but did not show whether the graph could be used for other purposes. Beeri and Bernstein [Beer79] used *derivation trees* to illustrate how a set X of attributes determines an attribute A that is not in X. Maier [Maie80] extended this tree to a general directed graph called a *derivation DAG* (Directed Acyclic Graph). This graph was only used to represent a derivation $X \rightarrow Y$, where X and Y are attribute sets. In both of these representations, vertices correspond to the attributes.

Hypergraph representations were used by Batini and D'Arti [Bati79] and Degano, Lamanto, and Sirovich [Dega80]. In both [Bati79] and [Dega80] attributes correspond to vertices and functional dependencies correspond to directed surfaces. The graphical representation of Batini et al. is similar to our graph representation, but our notation and operations on the graph are different, because we deal with problems related to logical design while they deal with query optimization problems.

Zaniolo and Melkanoff [Zani82] used directed multigraphs to represent one-to-one, one-to-many, and many-to-many relationships. In these multigraphs, the vertices are attributes and the edges are functional dependencies, labeled in order to identify the dependencies. The graph is used for finding a decomposition of the scheme.

Ausiello, D'Atri, and Sacca [Ausi83] used a directed graph to represent functional dependencies. There are two types of vertices: those representing one attribute and those representing a set of attributes. A compound vertex X has dotted directed edges to each attribute vertex A contained in X. Regular edges represent the functional dependencies. They gave graph algorithms for finding closures, and described properties such as redundancy and LR-minimality in terms of this graph representation.

Cosmadaki and Kanellakis [Cosm84] used graphs to represent functional and inclusion dependencies. Only unary functional dependencies are allowed, i.e., functional dependencies with single attribute left-hand sides. The vertices correspond to an attribute in a specific relation. There are red edges for functional dependencies and black edges for inclusion dependencies. The graph is used for solving implication problems.

Thus, until now, only Ausiello, D'Atri, and Sacca [Ausi83] have used graphs to manipulate functional dependencies. In the following sections we present two graph representations of functional dependencies and describe algorithms based on operations on these graphs.

3.3. The Semilattice Representation

For most functional dependency sets F, there are many nontrivial dependencies that can be inferred from F. Many of these dependencies, though difficult to discover, can be useful to the user when the semantics of the database is being specified. We would like to find a set of dependencies F' equivalent to F that contains all the nontrivial dependencies that can be derived from F. F' should be of such a form that all dependencies of F^+ can be generated simply by taking the union of two or more dependencies. To state this more formally, let us represent the functional dependencies over the attributes R as ordered pairs. The functional dependency $X \rightarrow Y$ is thus written as (X,Y) and the union of dependencies (X,Y) and (Z,W) is (XZ,YW). We focus on functional dependencies of the form (X,X^+), where X^+ is the closure of X with respect to F. Let S_F be a set of pairs (X,X^+) where $X \in 2^R$. We define the relation $\leq$ on S_F as:

$(X,Y) \leq (Z,W)$ iff $X \subseteq Z$ and $Y \subseteq W$.

This relation is *reflexive*, since $(X,Y) \leq (X,Y)$ is always true. The relation is also *transitive*, since $(X,Y) \leq (Z,W)$ & $(Z,W) \leq (V,U) \Rightarrow (X,Y) \leq (V,U)$ holds. *Antisymmetry* also holds, since $(X,Y) \leq (Z,W)$ & $(Z,W) \leq (X,Y) \Rightarrow (X,Y) = (Z,W)$. Thus, the relation $\leq$ over S_F is a *partial order*. Set S_F has an upper bound (R,R), where R is the set of all attributes. This means that we can form an upper semilattice of the set S_F.

We can graphically represent the functional dependencies as a semilattice, i.e., a graph in which the vertices correspond to members of S_F and the edges represent the partial order. There is a directed edge between vertices (X,Y) and (Z,W) when $(X,Y) \leq (Z,W)$ and there is no distinct vertex $(V,U) \in S_F$, such that $(X,Y) \leq (V,U) \leq (Z,W)$. We will display the graph so that vertex (X,Y) is at a lower level than vertex (Z,W) if $(X,Y) \leq (Z,W)$, so we can omit the arrowheads of the edges. We will also use the notation $(X,Y)_{i,j}$, where i is the level of element (X,Y) in the semilattice and $j \in [1, \binom{||R||}{i}]$ the element number in that level. This means that elements that are ordered less than $(X,Y)_{i,j}$ are on levels less than i.

Example 3.4: Consider the example $R(A,B,C)$ and $F = \{AB \rightarrow C, C \rightarrow B\}$. Because of the above definitions we, can order all possible functional dependencies of the form $X \rightarrow X^+$ (i.e., F^+) as illustrated in Figure 3.1. □

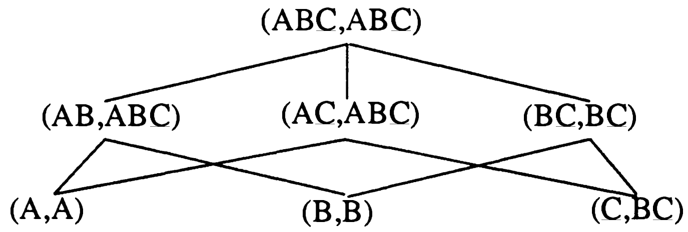


Figure 3.1. Semilattice Representation

The semilattice representation is large for most functional dependency sets. By excluding trivial dependencies, i.e., elements with identical left- and right-hand sides, and elements that are redundant with respect to lower ordered elements, the remaining set of functional dependencies will contribute to the set of functional dependencies that we call the *basis* $B(F)$.

Definition 3.12: *Basis* $B(F)$. The basis $B(F)$ of F^+ is built by the dependencies $(X,Y)_{i,j}$ from the semilattice S_F where $(X,Y)_{i,j}$ is:

1. $X \cap Y \neq \emptyset$ and
2. $(X,Y)_{i,j} \neq \cup_k (Z,W)_{i-1,k}$ where $k \in [1, \binom{||R||}{i}]$.

and with right-hand sides Y are reduced to $\hat{Y}$ with respect to lower ordered dependencies.

3. $\hat{Y} = Y - \cup\{W \mid (Z,W)_{l,k} < (X,Y)_{i,j} \text{and} (Z,W)_{l,k} \in B(F) \text{and} 1 \leq l,i\}$

In our previous example, $B(F)$ is the same as $F = \{AB \rightarrow C, C \rightarrow B\}$. All the dependencies of F^+ can be found by the union of dependencies in $B(F)$ and the trivial dependencies (A_i, A_i) where $A_i \in R$.

Example 3.5: Now consider again the example R={Teacher, Course, Hour, Room, Student, Grade}={T, C, H, R, S, G} with the functional dependencies $F=\{\ C \rightarrow T,\ CS \rightarrow G,\ HR \rightarrow C,\ HS \rightarrow R,\ TH \rightarrow R\}$. Part of the semilattice representation for this dependency set is given in Figure 3.2. The basis $B(F)$ is

$\{C \rightarrow T\ , CH \rightarrow R\ , CS \rightarrow G\ , TH \rightarrow RC\ , HR \rightarrow C\ , HS \rightarrow CGRT\}$

We see that $C \rightarrow T$ is the only nontrivial dependency on the lowest level. The pair (CS,$CTSG$) was first reduced to $CS \rightarrow TG$, and then T was removed since it was redundant with respect to dependency $C \rightarrow T$. The dependency $HS \rightarrow CGRT$ does not have any redundant dependencies with respect to lower level dependencies and thus it remains unchanged. □

The basis $B(F)$ contains redundant dependencies, but it also contains nontrivial dependencies that can be useful. In the example, the basis includes the nontrivial dependency $CH \rightarrow R$, which was not initially in F. The dependency set F is a cover and a subset of $B(F)$, i.e., the dependency $X \rightarrow Y$ is such that $X \rightarrow Z$ is in $B(F)$ and $Y \subseteq Z$. In addition, we find that the following subset of $B(F)$ is also a cover:

$\{C \rightarrow T\ , CH \rightarrow R\ , CS \rightarrow G\ , TH \rightarrow C\ , HR \rightarrow C\ , HS \rightarrow T\}$.

This cover could not be derived directly from F. Thus, the set $B(F)$ can be used to generate additional covers for F. In general, starting with $B(F)$, one might find more equivalent covers of F since $B(F)$ contains nontrivial dependencies implied from F but not contained in F.

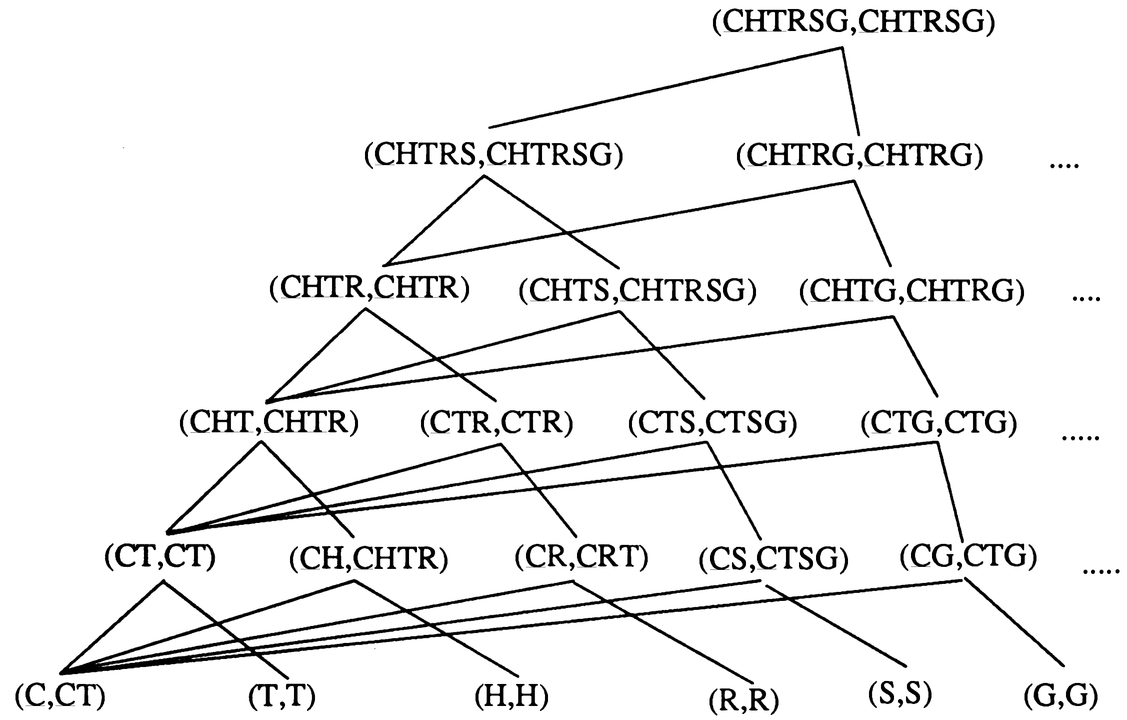


Figure 3.2. Semilattice Representation

One way to construct the basis $B(F)$ is to start building the semilattice representation bottom-up. First construct all (A_i, A_i^+) for the attributes $A_i \in R$, then $(A_iA_j, (A_iA_j)^+)$, $(A_iA_jA_k, (A_iA_jA_k)^+)$, and so on. $B(F)$ is found by selecting the pairs (X,Y) where X is a proper subset of Y and (X,Y) is not found by the union of some smaller ordered pairs. Then, attributes determined by lower ordered dependencies are removed from the right-hand side. Building the semilattice in this way is clearly exponential in the number of attributes, since the left-hand sides will form the power set 2^R. Thus, this approach is not efficient. In Section 3.7 we will give an efficient construction of the basis $B(F)$ and show that for some classes of functional dependency sets this construction takes only polynomial time.

3.4. The Attribute-Fd Graph

We propose a graph representation for functional dependencies that is an extension of an ordinary directed graph and allows us to characterize properties of functional dependencies in terms of graph properties. We are assuming that set F is in a form in which there is only one occurrence of a left-hand side, that is, dependencies with the same left-hand sides are grouped, and the right-hand side attributes are disjoint from the left-hand side attributes.

Definition 3.13: *The Attribute-Fd Graph.* For a given set of functional dependencies $F=\{f_1, f_2, ..., f_p\}$ over attributes $R=\{A_1, A_2, ..., A_m\}$, the attribute-fd graph $G_F=(\mathbf{V},\mathbf{E})$ is a directed graph with vertices $\mathbf{V}=R \cup F=\{A_1, A_2, ..., A_m, f_1, ..., f_p\}$ and directed edges $\mathbf{E}=\{A, X, Y : A \in R, A \in X, (X,Y) \in F : (A,(X,Y))\} \cup \{A, X, Y : A \in R, A \in Y, (X,Y) \in F : ((X,Y),A)\}$

The attribute-fd graph has two types of vertices, those associated with an attribute name and those associated with a functional dependency. The edges can only be between a functional dependency vertex and an attribute vertex.

The number of vertices in the graph is $\|\mathbf{V}\|=\|R\|+\|F\|$, where $\|F\|$ is the number of functional dependencies in F and $\|R\|$ is the number of attributes in R. The number of edges is given by the number of the attributes appearing on the left-hand side or the right-hand side of the dependencies, so $\|E\|=|F|$.

Example 3.6: Consider the functional dependency set $F=\{ C \rightarrow T, CS \rightarrow G, HR \rightarrow C, HS \rightarrow R, TH \rightarrow R\}$ of Example 3.5 . The attribute-fd graph of this dependency set is given in Figure 3.3. □

3.5. The Closure Algorithm

During the computation of covers of functional dependencies and other logical design algorithms, one often needs to know whether a dependency $X \rightarrow Y$ is implied by the dependency set F. This problem is called the *membership problem*, since it determines whether $X \rightarrow Y \in F^+$. Clearly, one should avoid computing the closure F^+ to answer the membership problem. The problem can also be solved by finding the attributes that functionally depend on X, that is the closure X^+. Finding X^+ can be solved

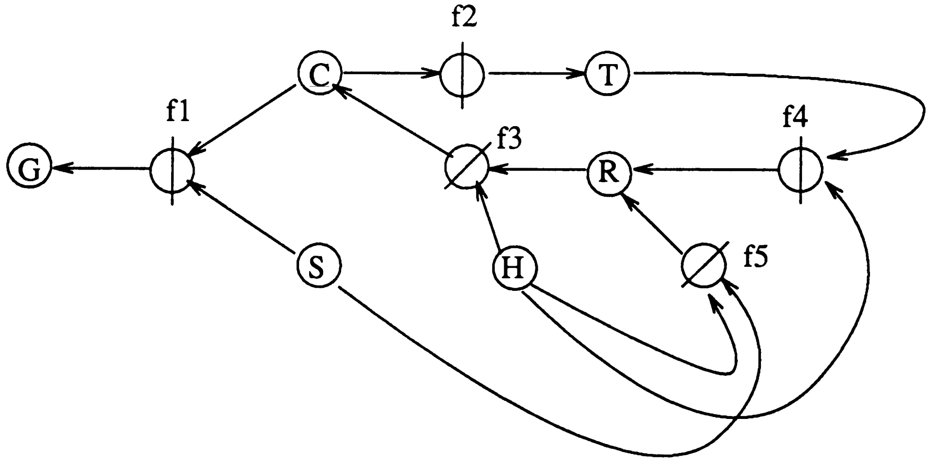


Figure 3.3. Attribute-Fd Graph

by using F without knowing F^+. The naive approach would be to find all dependencies with left-hand side containing attributes that are in X. Then X is augmented with the right-hand side attributes of these dependencies and this containment search is continued on the remaining dependencies until no left-hand side contains X. The resulting X will be X^+. This method is of O($\|R\| \cdot |F|$) time [Beer79], where $\|R\|$ is the number of attributes in the relation and $|F|$ is the number of attributes occurring in F or the length to represent F.

An O($|F|$) time algorithm to find X^+ is presented by Beeri and Bernstein [Beer79]. This algorithm uses the same principle as the previous method, but it is improved by keeping track of which dependency can be applied next. This is described in Algorithm 3.1.

To show that using the attribute-fd graph is as efficient as the previous approaches we must show that the closure of an attribute set can be found in linear time. We will also show that the use of the attribute-fd graph is easy to understand and provides a simple closure algorithm.

Our attribute-fd graph is restricted to handle reduced functional dependency sets defined as:

Definition 3.14: *Reduced Functional Dependency Set* [Ausi83]. A set F of functional dependencies on R is in *reduced form* if

1. $\forall X \rightarrow Y, X \cap Y = \emptyset$.
2. Left-hand sides are unique.

To find a cover in reduced form will take O($|F|^2$) time. Removal of attributes from the right-hand side that also occur on the left-hand side is a O($|F|^2$) time operation in the

Algorithm 3.1. Determine Closure X^+ of a Set of Attributes X [Beer79]

Input: A set of attributes X and a set of functional dependencies F.
Output The closure X^+

$\{f_1, f_2, ..., f_{||F||}\}$ are the functional dependencies.
$\{A_1, A_2, ..., A_n\}$ are the attributes.
Closure contains the closure of set X at termination.
AddedAttr is a subset of *Closure* where each element is added exactly once.
Attrlist[*i*] is a list of functional dependencies whose left sides contain A_i.
Counter[*j*] is the number of attributes in f_j.*lhs* not in *Closure–AddedAttr*.

1. {Initialization of *Attrlist*, *Counter*, *Closure* and *AddedAttr*}
    for *i*= 1 **to** *n* **do**
        Attrlist[*i*]:=NIL;
        for j= 1 **to** ||*F*|| **do**
            if $A_i \in f_j$.*lhs* **then**
                Attrlist[*i*]:=*Attrlist*[*i*]$\cup$\{*j*\};
    for j= 1 **to** ||*F*|| **do**
        Counter[*j*]:=||f_j.*lhs*||;
    Closure, AddedAttr:=X, X;

2. {For each A_i in *Closure*, decrease *Counter*[*i*] for all f_j containing A_i in the left side and add right side to closure whenever *Counter*[*j*]=0.}
    while (*AddedAttr*<>$\emptyset$) **do**
        AddedAttr:=AddedAttr–A_i;
        for each $f_j \in$ *Attrlist*[*i*] **do**
            Counter[*j*]:=*Counter*[*j*]–1;
            if *Counter*[*j*]= 0 **then**
                AddedAttr:=AddedAttr$\cup$(f_j.*rhs* –*Closure*);
                Closure:=Closure$\cup f_j$.*rhs*;

3. {Closure contains X^+}
    return(*Closure*)

worst case. To union the dependencies with the same left-hand side is also a $O(|F|^2)$ time operation. Since the complexity of transforming the dependency set into a reduced form is so high, it seems more reasonable to use a canonical cover (Definition 3.6) rather than a reduced functional dependency set, or better yet a reduced canonical cover defined as:

Definition 3.15: *Reduced Canonical Cover.* A set of functional dependencies F is a reduced canonical cover if F is nonredundant and for every $X \rightarrow Y \in F$

1. $X \rightarrow Y$ is left and right reduced and
2. No other dependency has left-hand side X.

With a reduced canonical cover we see that only two of the Armstrong axioms can be used to perform operations on the attribute-fd graph; the transitivity and the pseudo-transitivity rules. The reflexitivity rule cannot be applied because functional dependencies in a reduced cover have disjoint left- and right-hand sides. Similarly, augmentation cannot be used because it will form dependencies with extraneous attributes, since the left- and right-hand sides of a dependency will no longer be disjoint. The union rule is not applicable since two or more dependencies with the same left-hand side are not allowed. The decomposition rule cannot be used because its application will produce a violation of the reduced form. The transitivity and pseudotransitivity rules will correspond to a graph traversal along and against the directed edges of the attribute-fd graph. The transitivity rule will result in finding the closure of each left-hand side of a functional dependency. The pseudotransitivity rule will be used to find left-hand sides that can replace the left-hand sides initially in F, as we shall see in the next section.

Before describing our algorithms to find the closure of a set of attributes X, we introduce some notation and terminology. We say that a functional dependency vertex f_i is *dependent* on its predecessor (attribute) vertices or its left-hand side vertices. We denote this set of attributes by $f_i.lhs$. Similarly, we say that f_i is the *determinant* of its successor vertices or its right-hand side vertices, denoted by $f_i.rhs$. A functional dependency vertex will have predecessor and successor degrees corresponding to the number of predecessor and successor attribute nodes, denoted by $f_i.pre\_degree$ and $f_i.suc\_degree$. For a specific attribute A, we denote the dependencies containing A in the left-hand side for successors of A by $A.suc$. The dependencies containing A in the right-hand side we call predecessors of A, denoted by $A.pre$.

The closure of attributes X using the attribute-fd graph can be found by traversing the derivation path defined as follows:

Definition 3.16: *Derivation Path.* There is a *derivation path* $X \xrightarrow{+} A$ in the attribute-fd graph G_F if one of the following paths exist:

1. Direct path: (f_i, A), where $f_i.lhs \subseteq X$.
2. Indirect path: $(f_{k_1}, B_1)(f_{k_2}, B_2)$, ..., (f_{k_k}, B_k), where $f_{k_1}.lhs \subseteq X$ and $f_{k_i}.lhs \cup B_i \subseteq f_{k_{i+1}}.lhs$.

Using this definition, we can construct an algorithm to find the closure of a set of attributes X. The method marks the attribute vertices of X, then visits the functional dependency vertices with all the left-hand side attributes marked and successively marks the attributes that these dependency vertices determine. The algorithm terminates when there are no more functional dependency vertices that can be traversed. The algorithm is given in Algorithm 3.2.

Theorem 3.1: Algorithm 3.2 has complexity O(|*F*|).

Proof: The time to construct the attribute-fd graph for F is O(|*F*|). The algorithm requires that the left- and right-hand sides of a dependency are disjoint; however, F need not be a reduced canonical cover. Each dependency is added at most once to *apply_list*; when all the left-hand side attributes of the dependency have been marked (f_i*.mark_degree* = f_i*.pre_degree*) the dependency is added to *apply_list*. Each incoming and outgoing edge of a dependency vertex in the *apply_list* is manipulated only once. □

The algorithm has the same complexity as the linear algorithm presented in Algorithm 3.1. The major difference is that our algorithm traverses the attribute-fd graph rather than linked lists. The attribute-fd graph gives a better view of the dependency set than array *Attrlist* of Algorithm 3.1. Variable *Counter*[*j*] of Algorithm 3.1 and f_j*.mark-degree* of Algorithm 3.2 serve the same purpose of being used to decide when an attribute set X is a successor of a dependency f_j.

To build an attribute-fd graph whose dependencies point to all the closure attributes of its left-hand side attributes (equivalent to the closure graph in [Ausi83]), the above algorithm is modified by replacing statement $X^+ := X^+ \cup A$ with **if** $A \notin f_j.rhs$ **then** $f_j.rhs := f_j.rhs \cup A$. With this modification, the algorithm should be called for each functional dependency. Finding this graph takes O(||*F*||·|*F*|) time.

3.6. The Projection Algorithm

The next algorithm we present computes the projection of a set of functional dependencies F over R to a set of dependencies over $R' \subseteq R$. The projection of a functional dependency set is formally defined as:

Definition 3.17: *Projection of Functional Dependencies*. The projection of the functional dependencies F over R onto $R' \subseteq$R is the functional dependency set: $F_R' = \{X \rightarrow Y \mid X \rightarrow Y \in F^+, X \subseteq R'$ and $Y \subseteq R'\}$

The functional dependencies in F_R' are all induced by F. We are interested in an algorithm that will produce a projected set of functional dependencies that does not contain unnecessary or redundant dependencies. Using results from [Gott87], we can remove an attribute A from the left-hand side of a functional dependency by replacing it with attributes Y where $Y \rightarrow A \in F^+$. If the dependency set contains left-hand sides $Y_1, Y_2, ..., Y_k$

Algorithm 3.2. Determine Closure X^+ of a Set of Attributes X

Input: A set of attributes X and a set of functional dependencies F.
Output: The closure X^+.

Closure contains the closure of set X at termination.
apply_list is a list dependencies with left-hand sides in *Closure.*

0. {Create the attribute-fd graph G_F with all f_j*.mark_degree*=0.}

1. {Mark all dependencies adjacent to the attributes of X}
    apply_list:=∅;
    Closure:=X;
    for all $A \in X$ **do**
        FindUnappliedSuc(A);
2. {Traverse the functional dependencies whose left-hand sides are in *Closure* and add their right-hand sides to *Closure*}
    while (*apply_list* <> ∅) **do**
        remove *f from apply_list*;
        for all $A \in$ *f.rhs* **do**
            Closure:=*Closure* ∪ A;
            FindUnappliedSuc(A);

3. {Closure contains X^+}
    return(*Closure*)

procedure FindUnappliedSuc(A);
{For all f that succeed A, increment *f.mark_degree* and append f to *apply_list* if current *Closure* contains the left-hand side.}
    for all $f \in$ *A.suc* **do**
        f.mark_degree:=*f.mark_degree*+1;
        if (*f.mark_degree* =*f.pre_degree*) **then**
            apply_list:=*apply_list*∪*f*

that all determine A, and left-hand sides $X_1A, X_2A, \ldots, X_pA$ containing A, we will introduce the left-hand sides: X_1Y_1, $X_1Y_2, \ldots$, X_1Y_k, X_2Y_1, $X_2Y_2, \ldots$, X_2Y_k, ..., X_pY_1, $X_pY_2, \ldots$, X_pY_k. Thus, a projection that removes one attribute A from R will introduce at most *A.pre_degree*A.suc_degree* new dependencies, if F is in reduced form.

Use of the attribute-fd graph enables us to select the dependencies that can possibly produce new dependencies. Gottlob [Gott87] gives a similar algorithm, based on using the transitivity rule for finding projections of functional dependency sets, as described above. The complexity of his algorithm is $O(\|R\|^5 \cdot |F|^{3\|R\|})$. However, in his method all the dependencies are tried; they are not grouped logically as in the attribute-fd graph. Thus, the graph algorithmic approach makes it possible to restrict manipulation to the dependencies that will be applied, and the other dependencies will not be considered. The projection algorithm is given in Algorithm 3.3.

Algorithm 3.3. Project Dependency Set F over Attributes R onto $R-A$

Input: Attributes R, functional dependencies F, and attribute A.

Output: The projection F_{R-A}.

update_list is a set of dependencies containing A.

1. {Create new dependency $XY \rightarrow B$ from dependencies $XA \rightarrow B$ and $Y \rightarrow A$ }
    update_list:=$\emptyset$;
    for each $f_j \in$ *A.suc* **do**
        for each $f_k \in$ *A.pre* **do**
            $F := F \cup \{(f_j.lhs \cup f_k.lhs - A) \rightarrow f_j.rhs\}$;
            update_list:=*update_list* $\cup\, f_k$;
        <remove f_j and all its lhs and rhs edges.>
2. {Remove A from the remaining dependencies }
    for each $f_j \in$ *update_list* **do**
        $f_j.rhs := f_j.rhs - A$;
        if $f_j.rhs = \emptyset$ **then**
            <remove f_j and all its lhs edges.>

Theorem 3.2: The time for projecting over R onto $R-A$ is $O(\|R\| \cdot |F|^2)$.

Proof: In Algorithm 3.3, step 1 will at most loop through $|F|^2$ dependencies. Creating the new dependency and adding it to the dependency set takes $O(\|R\|)$ steps. The removal of a dependency is also $O(\|R\|)$ time. This means that step 1 takes $O(\|R\| \cdot |F|^2)$ time. Step 2 of the algorithm is clearly $O(\|R\| \cdot |F|)$ time. Thus, the algorithm is $O(\|R\| \cdot |F|^2)$ time. □

To find the projection of a dependency set F onto any set of attributes $R_i \subseteq R$, we repeatedly use Algorithm 3.3, called *project(F,R,A)* and we get the generalized projection algorithm given in Algorithm 3.4. Each time *project(F,R,A)* is called, some new dependencies are introduced and some removed. To find the time complexity of the generalized projection algorithm, these dependencies must be taken into account.

Theorem 3.3: The time complexity of Algorithm 3.4., projecting F over R onto $R_i \subseteq R$, is of $O(\|R\|^2 \cdot 2^{2\|R\|})$.

Proof: Let u be an upper bound for the cardinality of the functional dependencies at any time during the execution. Then, following Theorem 3.2, the time complexity is $O(\|R\|^2 \cdot \|u\|^2)$. There cannot be more than $2^{\|R\|}$ possible left-hand sides, which is also the size of u. This gives that the algorithm is of $O(\|R\|^2 \cdot 2^{2\|R\|})$. □

3.7. Finding the Basis $B(F)$

To find all the dependencies of the basis $B(F)$ described in Section 3.3, we will use the projection algorithm and a reduced canonical cover. The possible left-hand sides of

---

Algorithm 3.4. Project Dependency Set F onto Attributes R_i

Input: Attribute sets R, R_i, and functional dependencies F on R.

Output: The projection F_{R_i}.

for each $A \in R-R_i$ **do**
    project(F,R,A);
    $R=R-A$

F^+ with no extraneous attributes can be found by modifying the algorithm to left reduce the introduced dependencies. The algorithm must also be modified to not remove dependencies. The projection part of the algorithm is performed on all attributes $A \in R$ and not only on $A \in R-R_i$. The algorithm must also apply the transitivity rule to introduced dependencies not handled through the regular substitution process. Each dependency is handled once, and the method will eventually terminate, since we know that the size of $B(F)$ is bounded by the size of F^+. Clearly the added dependencies are redundant with respect to F, but these dependencies will not be redundant with respect to some projection. For a class of dependency sets, we will show that the size of basis $B(F)$ will be considerably less than $2^{\|R\|}$. This class is a generalization of a class of functional dependencies recently defined by Gottlob.

Definition 3.18: *Class H* [Gott87]. The class H is a set of functional dependency sets such that for any F in H, F satisfies the condition: For each pair of dependencies $X \rightarrow A$ and $Y \rightarrow B$ in F, A is not in Y.

Gottlob showed that for functional dependency sets from class H his projection algorithm takes polynomial time. H contains dependency sets where attributes that are in the right hand side of a dependency cannot be contained in a left-hand side. Thus, if the attribute is in a left-hand side it is the only attribute in that left-hand side. The projected dependency set will not contain any new left-hand sides. This means that $B(F)$ will have the same number of dependencies as F, but it can contain more attributes, that is $\|B(F)\|=\|F\|$ and $|B(F)|\geq|F|$.

Let us say that $F_1 \subseteq F$ are the dependencies that have accrued some extra attributes in the right-hand side during the computation of $B(F)$. Gottlob's class H has restricted the set F_1 to contain dependencies with only single attribute left-hand side. We would like to extend this class so that it can also contain dependencies with left-hand sides of arbitrary size.

Example 3.7: Consider the functional dependency set $F=\{DE \rightarrow AB,\ AB \rightarrow C\}$ from Example 3.5 . This dependency set does not belong to H, since $A \subset AB$. But $B(F)=\{DE \rightarrow ABC,\ AB \rightarrow C\}$ has the same number of dependencies as F. □

We define a new dependency set class:

Definition 3.19: *Class* $\hat{H}$. The class $\hat{H}$ is a set of functional dependency sets such that for reduced canonical cover $F \in \hat{H}$, F satisfies the condition: For all pairs of dependencies $X_i \rightarrow Y_i$ and $X_j \rightarrow Y_j$ in F, we have that $X_j - Y_i = \emptyset$ or $X_j - Y_j = X_j$.

The class $\hat{H}$ will include dependency sets similar to the one given in Example 3.3.

3.8. The Key Finding Algorithm

Lucchesi and Osborn [Lucc78] presented an efficient key finding algorithm. The algorithm is based on finding one key, say K_1, and adding it to a set S of keys. Then

$K = K_j X - Y$, where $X \rightarrow Y \in F$ and $K_j \in S$ is a candidate key. To determine whether it can give a new minimal key, K must be neither a member nor a subset of a member of S. If these conditions hold, then extraneous attributes are removed from K and the resulting set is added to S. These operations are performed on all the minimal keys found (see Algorithm 3.5).

An initial key is any set X that determines all the attributes R. This set is found by removing any extraneous attributes from the left-hand side of the dependency $X \rightarrow R$, where X is initially set to R. Then X is set to $X-A$ if $R-A \rightarrow A \in F^+$. The remaining set of

---

Algorithm 3.5. Determine All the Minimal Keys of R Given F [Lucc78]

Input: A set of attributes R, a set of functional dependencies F.

Output: The set of minimal keys.

K is a set of minimal keys
candidate_key is an attribute set that might contain a new key.
new_key is set to false if *candidate_key* contains no new minimal key.

```
1. {Find one minimal key K1}
      K:=find_one_key(R);
2. {Find all the minimal keys}
      for each Ki ∈ K do
            for each X→Y∈F do
                  candidate_key:=KiX− Y;
                  new_key:=true;
                  for each Kj∈ K do
                        if candidate_key⊇Kj then new_key:=false;
                  if new_key then K:=K∪ find_one_key(candidate_key);

procedure find_one_key(X)
{Find a minimal key that is a subset of X}
```

---

attributes X is a key [Lucc78].

The disadvantage with the above algorithm is that all functional dependencies are examined in step 2. Clearly, no new key will be found from key K_j and dependency $X \rightarrow Y$ if $X \cap K_j = \emptyset$. This case can happen often in practice. Consider the dependency set $F = \{CS \rightarrow G, C \rightarrow T, HR \rightarrow C, HT \rightarrow R, HS \rightarrow R\}$ given in Figure 3.3. The initial key will be $K_1 = \{HS\}$. Using the attribute-fd graph of F, there is no need to try all dependencies to determine whether $K = K_1X - Y$ is a minimal key. This is apparent because no dependency in F determines an attribute of $K_1 = \{HS\}$ and thus K_1 will be contained in all $|F|$ candidate keys found by Lucchesi's and Osborn's algorithm. Thus given a key K_1, we only need to use dependencies $X \rightarrow Y$ where $K_1 \cap Y \neq \emptyset$. In the attribute-fd graph these dependencies correspond to the dependency nodes adjacent to the attributes of K_1. Use of the attribute-fd graph avoids some of the unnecessary testing performed in the conventional method. In addition, we have the following theorem:

Theorem 3.4: Let K_1 be minimal key of R with functional dependencies F. If F is LR-minimal and $S = \{K_1X - Y \mid X \rightarrow Y \in F \text{ and } K_1 \cap Y \neq \emptyset\}$, then for any $S_j \in S$ there is no other $S_i \in S$, $i \neq j$, such that $S_j \subseteq S_i$.

Proof: Consider dependencies $X_i \rightarrow Y_i$ and $X_j \rightarrow Y_j$ of F. Clearly, $K_1X_i - Y_i \not\subseteq K_1X_j - Y_j$ if $X_i \not\subseteq X_j$. Now, if $X_i \subset X_j$ then we know that $Y_i \cap Y_j = \emptyset$ since F is LR-minimum. This implies that $K_1X_i - Y_i \not\subseteq K_1X_j - Y_j$ since we only search for new keys in the case $K_1 \cap Y_i \neq \emptyset$. Thus, no element of S contains another element of S. □

The above theorem gives another possible optimization to the key finding algorithm. It shows that one can compute S first and then remove elements from S that are supersets of or equal to previously found keys.

3.9. Algorithm for Finding the Generator

An Armstrong relation is a special relation that can show both the existence and the nonexistence of dependencies for a relation. That is, a dependency holds for the relation if and only if it holds for the Armstrong relation. Thus, an Armstrong relation is the dual way of representing or showing a dependency set. The use of Armstrong relations in database design was proposed by Silva and Melkanoff [Silv81] and Bitton, Mannila, and Räihä [Bitt85]. Unfortunately, the Armstrong relations have a drawback compared to the conventional linear representation of a dependency set. It has been shown [Beer84b] that for some dependency sets, the size of all Armstrong relations must be exponential both in the number of attributes and in the number of dependencies. However, these relations were highly unnormalized. For normalized relations the situation is quite different. Mannila and Räihä [Mann85] have shown that the size of an Armstrong relation for schemes in BCNF depends only on the number of keys in the relation. They showed that the size of a minimal Armstrong relation for a relation in BCNF is exponential only in the number of dependencies (i.e., the number of keys) and not in the number of attributes. Since the number of keys is usually small, this provides a lower bound for the size of an Armstrong relation. A similar bound is found for relations in 3NF.

Even the best algorithms for finding Armstrong relations suffer from efficiency problems when the relation is unnormalized. In the next section, we propose a different approach to finding Armstrong relations that might be faster in cases that frequently occur during the logical design process.

Armstrong relations can be regarded as a representation of the closure of a dependency set. Two tuples in a relation instance can only agree on attributes that are in the closure of some attribute set. Thus, one can view the problem of finding an Armstrong relation as finding the attribute sets that any two tuples in the relation can agree on without violating any of the given dependencies. Given a set of attribute sets, S, we can find the Armstrong relation as described in Algorithm 3.6.

Before we describe an algorithm to find a family of attribute sets that can be used to generate an Armstrong relation, we give some definitions.

Definition 3.20: *F-Closed and CL(F)* [Beer84b]. Let R be a relation scheme and F a set of functional dependencies over R. A set X subset of R is *F-closed* if for all A in R we have $X \rightarrow A \in F^+$ if and only if $A \in X$. The set of all *F-closed* sets of R is denoted by $CL(F)$.

Since $CL(F)$ is closed under intersection [Arms74], there exists a unique minimal subfamily of *generators* $GEN(F)$ [Beer84b] that is a subset of $CL(F)$. Each member of $CL(F)$ can be expressed as an intersection of sets in $GEN(F)$. It is clear that $GEN(F)$ must contain the "largest" sets in $CL(F)$ since there is no way they can be found by intersection of other sets. Thus, we can select the sets Y from $CL(F)$ that have the following characteristic: If some attribute B in $R-Y$ is added to Y then $YB \rightarrow A$ for some A in $R-YB$, and no set properly containing Y (but not A and B), has this property. This is stated more clearly in the following theorem.

Theorem 3.5: [Mann85] $GEN(F) = MAX(F) = \cup_{A \in R} max(F,A)$, where $max(F,A) = \{Y \subseteq R \mid Y \neq \emptyset$ and maximal such that $Y \rightarrow A \notin F^+\}$.

It is desirable to have small Armstrong relations, and it turns out that $GEN(F)$ is the smallest attribute set that the tuples can agree in a relation r and still satisfy the given dependencies. An appropriate set S for Algorithm 3.6 should then be a set such that $GEN(F) \subseteq S \subseteq CL(F)$. Thus, one should use $GEN(F)$ to construct an Armstrong relation, or rather $MAX(F)$, since $MAX(F)$ gives a more constructive way of finding $GEN(F)$. We refer to [Mann85] for more details.

$MAX(F)$ can be computed incrementally, starting with $max(G,A)$ for $G=\emptyset$ and successively adding a functional dependency at a time to G and computing $MAX(G)$ until $G=F$. Initially $max(\emptyset,A)=R-A$ for all $A \in R$. The following result from [Mann85] shows how $max(G,A)$ relates to $max(F,A)$ when G contains one more functional

Algorithm 3.6. Construct an Armstrong Relation **r** [Beer84b]

Input: Sets of attributes R and S.
Output: A relation r over R.

tuple set of attribute values
i integer counter

```
1. {initialize relation r}
     for each A∈R do tuple[A]:=0;
     insert(tuple,r);
     i:=0;
2. {Create the desired tuples form S}
     for each T∈S
          for each A∈R do
               if each A∈T
                    then tuple[A]:=0;
                    else tuple[A]:=i;
          insert(tuple,r);
          i:=i+1;
```

dependency than F.

Theorem 3.6: [Mann85] Let $G=F\cup\{Y\rightarrow Y'\}$ and $W\in max(G,A)$. Either $W\in max(F,A)$, or for some $B\in Y$, $Z\in max(F,B)$ and $X\in max(F,A)$ we have $W=X\cap Z$.

This theorem gives an incremental construction of $GEN(G)$. In an algorithm based on this approach, one must repeatedly check whether some potential W really is maximal; this check requires time $O((\|R\|+1)\cdot|F|)$. The disadvantage of this method is that some intermediate set of dependencies can have exponential size. Consider the two dependency sets

$F=\{A_{2i-1}A_{2i}\rightarrow B \mid 1 \le i \le m\}$ and
$G=\{A_iA_j\rightarrow B \mid 1 \le i < j \le 2m\}$

We see that $F\subset G$, since left-hand sides of dependencies in F are a selection of the left-hand sides of dependencies in G. The size of $GEN(F)$ is exponential, since $max(F,B)$ is all the combinations of m distinct attributes from the set of $2m$ attributes excluding those sets that contain a left-hand side of dependencies in F. Since the left-hand sides of the dependencies of F are disjoint, an element of $max(F,B)$ can be found by selecting one attribute from each left-hand side of F and removing them from R. That is, select A_{2i-1} or A_{2i} for each i, i = 1,...,m and remove those attributes from R. The cardinality of $max(F,B)$ is the number of possible combinations found by selecting one attribute from m sets of size two, or 2^m. A similar analysis shows that $GEN(G)$ is of polynomial size.

Note that since $F\subset G$, computing $GEN(G)$ can be exponential in the number of attributes even if the final result, $GEN(G)$, is polynomial. At some intermediate point the method can have computed $GEN(F)$ that has exponential size.

We propose a different approach for finding $max(F,A)$, based on the following notion of minimal sets:

$min(F,A)=\{Y\subseteq R \mid Y\neq\emptyset$ and minimal such that $Y \rightarrow A\}$.

Set $min(F,A)$ contains all the sets of attributes Y that can determine A and where there are no extraneous attributes in Y. By definition, $A\in min(F,A)$. Using this $min(F,A)$, we can generate $max(F,A)$. The following lemma relates $min(F,A)$ and $max(F,A)$:

Lemma 3.7: Given $min(F,A)$ and $max(F,A)$ of a functional dependency set F and an attribute $A\in R$, $Y_i\not\subseteq Z_j$ holds for all $Y_i\in min(F,A)$ and for all $Z_j\in max(F,A)$.

Proof: Clearly $Y_i\not\subseteq Z_j$ is true; otherwise from $Y_i\rightarrow$A it would follow that $Z_j\rightarrow$ A, which contradicts that Z_j is a member of $max(F,A)$. □

Given this Lemma, it follows that the complement of each elements of $min(F,A)$ must intersect with all elements of $max(F,A)$. We can use this property to generate the maximal sets from the minimal sets, since this property resembles the notion of *hitting sets* of the minimal sets. The hitting sets of S are defined as:

$hs(S)=\{Y\subseteq R \mid \forall(X\in S, Y\cap X\neq\emptyset)\}$.

Not all the elements of $hs(min(F,A))$ can be used to produce $max(F,A)$, since there are some members of $hs(min(F,A))$ that are supersets of other members, if we can remove these members we will find the set containing the complement sets of the maximal sets. This set with supersets removed can be defined as:

$sr(S)=\{X\in S \mid \forall\ Y\in S, Y\subseteq X$ only if $Y=X\}$.

Using this notation we have the following theorem:

Theorem 3.8: $max(F,A)=\{R-X \mid X\in sr(hs(min(F,A)))\}$.

Proof: (i) Suppose $X\in sr(hs(min(F,A)))$. We must show that $R-X\to A \notin F^+$. Clearly, $R-X \not\supseteq Z_i$ for all $Z_i\in min(F,A)$, since $X\cap Z_i\neq\emptyset$ for all $Z_i\in min(F,A)$, but then $R-X\to A \notin F^+$. It remains to show that $R-X$ is maximal. If we add any $X_i\in X$ to $R-X$ we find that $(R-X)X_i\to A\in F^+$ since $\exists\, Z_i\in min(F,A)$ such that $(R-X)X_i\supseteq Z_i$. If there is no such Z_i contradicts the fact that $sr(hs(min(F,A)))$ contains no element that is a superset of another element. Thus, we have showed that $max(F,A)\subseteq\{R-X \mid X\in sr(hs(min(F,A)))\}$. (ii) Suppose $Y\in max(F,A)$. From Lemma 3.7 we know that $Y \not\supseteq Z_i$, for all $Z_i\in min(F,A)$. But then $R-Y\in hs(min(F,A))$ since $R-Y$ intersects with all the members of $min(F,A)$. By definition of $max(F,A)$, we know that there is no $Y'\subseteq Y$ that is also maximal, thus $R-Y\in sr(hs(min(F,A)))$. Thus we have showed that $max(F,A)\supseteq\{R-X \mid X\in sr(hs(min(F,A)))\}$. □

Example 3.8: Let us consider $\Sigma=\{AB\to G,\ CD\to G,\ EF\to G\}$, from this we get that

$min(\Sigma,A)=\{A\}$, $min(\Sigma,B)=\{B\}$,
$min(\Sigma,C)=\{C\}$, $min(\Sigma,D)=\{D\}$,
$min(\Sigma,E)=\{E\}$, $min(\Sigma,F)=\{G\}$,
$min(\Sigma,G)=\{G,AB,CD,EF\}$.

It is simple to see that $max(\Sigma,A)=\{R-A\}=\{BCDEFG\}$. Similarly, we can find the maximal sets for B, C, D, and E. To generate a subset of the hitting set that will contain all the complements of $max(\Sigma,G)$, we will select one attribute from each member of $min(\Sigma,G)$. If an attribute of a set is already selected, no selection is done and one continues to next member. Remove any supersets generated. In this example, the generation is rather simple and we get $sr(hs(min(\Sigma,G)))=\{GACE,\ GACF,\ GADE,\ GADF,\ GBCE,\ GBCF,\ GBDE,\ GBDF\}$ Note that in this case $sr(hs(min(\Sigma,G)))=hs(min(\Sigma,G))$. From this set, we can find the maximal sets by the complement of each member giving $max(\Sigma,G)=\{BDF,\ BDE,\ BCF,\ BCE,\ ADF,\ ADE,\ ACF,\ ACE\}$ □

It will not always be so simple to generate the maximal sets from $sr(hs(min(F,A)))$. For some attributes, it is often the case that the members will intersect as we shall see in the following example.

Example 3.9: Consider $\Sigma=\{AB\to G,\ BC\to G,\ CD\to G,\ DE\to G,\ EF\to G\}$, the minimal sets for A, B, C, D, E, and F are the same as in Example 3.8 . Thus, the maximal sets also are. But in this case we have $min(\Sigma,G)=\{G,AB,BC,CD,DE,EF\}$ Generating the hitting set by selecting one attribute from each element of $min(\Sigma,G)$ we get that $hs(min(\Sigma,G))=\{GABCE,\ GABCDF,\ GABDE,\ GABCDF,\ GACDE,\ GACDF,\ GABCE,\ GBCDE,\ GBCE,\ GBCF,\ GBCDF,\ GBCE,\ GBCF,\ GBDE,\ GBDF\}$. We see that this set contains many supersets. Removing the supersets we get $sr(hs(min(\Sigma,G)))=\{GACE,$

$GACF, GBCE, GBCF, GBDE, GBDF\}$ □

In this case we see that it would be better to remove the supersets while we are finding the hitting sets, instead of removing supersets in a separate step. But we can, of course, still get a rather large intermediate set. The method is described in more detail in Algorithm 3.7.

Note that $min(F,A)$ can be generated from the basis $B(F)$ that contains all non-trivial dependencies of F^+. It turns out that all left-hand sides of dependencies that contain A on the right-hand side will be in $min(F,A)$; A is added to $min(F,A)$, as well.

3.10. Finding Decompositions in Boyce-Codd Normal

One of the major tasks involved in logical design is finding normalized decompositions for a specified set of dependencies. A database design tool should be able to present alternative design possibilities, giving the database designer a choice, rather than finding only one decomposition.

Below, we present a way to find several decompositions in BCNF based on the basis $B(F)$ of a functional dependency set F. The method is based on Tsou's and Fischer's polynomial algorithm [Tsou82] for finding a decomposition in BCNF. The algorithm is described below in Algorithm 3.8. The main idea behind this algorithm is finding a minimal set $T \subseteq S$ that will be in BCNF, which is assured if for all $A,B \in T$ we have $T-AB \rightarrow A \notin F^+$. Thus, all the keys of T are of size $|T|-1$ (if not some attribute of T is removed). The relation $R_i=T$ is created and the process is repeated on $S=S-A$ until S has only two attributes or $T=S$. Rather than performing all the pruning that is done in the Fisher-Tsou algorithm we will predict the relations that can be found using this algorithm.

In the Fisher-Tsou algorithm, the last attribute A handled is removed from S and the pruning is repeated on this new S as mentioned. The possible sets that can minimally determine A can be found by inspecting basis $B(F)$. Clearly, all minimal left-hand sides are exactly those found by listing all dependencies of $B(F)$ that contain A on their right-hand side. Thus, by predicting which attribute will be removed from S, one can give the possible relations that can occur. Projecting the remaining dependencies onto $S-A$, we can repeat the procedure as in the Fisher-Tsou algorithm. In a sense, we are trying out all possible orders of attribute removal. Let us illustrate the method with the following example.

Example 3.10: Consider the relation $R=ABC$ and functional dependencies $F=\{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$. The basis is $B(F)=\{A \rightarrow B, A \rightarrow C, B \rightarrow C, B \rightarrow A, C \rightarrow A, C \rightarrow B\}$. Thus, we have three possible selection orders. These possibilities are found first by removing A, giving the relations AC and AB, found from the dependencies $C \rightarrow A$ and $B \rightarrow A$, respectively. Since the remaining set $S=BC$ is of size two, and thus in BCNF, we can halt the search. This gives the decompositions $\{AB,BC\}$ and $\{AC,BC\}$. But we need to also start removal of another attribute, say B. Following the same procedure as before, we find

Algorithm 3.7. Find $GEN(F)$

Input: A set of attributes R, a set of functional dependencies F.
Output: $GEN(F)$

1. {Find the basis $B(F)$}
2. {Generate $min(F,A)$ for all $A \in R$}
    for each $X \rightarrow Y \in B(F)$ **do**
        for each $A \in Y$ **do** $min(F,A) := min(F,A) \cup X$;
3. {Find the $max(F,A)$ for all $A \in R$}
    for each $A \in Y$ **do** search_max($1, min(F,A), max(F,A), \varnothing$);
4. {Produce the generator $GEN(F)$}
    for each $A \in Y$ **do** $GEN(F) := GEN(F) \cup max(F,A)$;

procedure search_max(i, *min*, *max*, *cur_set*)
  {procedure finds *max* set from $min = \{S_1, S_2, \ldots, S_k\}$}
    if $i = |min|$ **then** {Base Case}
        if ($S_i \cap cur\_set \neq \varnothing$) **then**
            add_and_remove_supersets(*max*,*cur_set*);
        else for *each* $C \in S_i$ **and** $C \notin cur\_set$ **do**
            add_and_remove_supersets(*max*,$cur\_set \cup C$);
    else {Recursive Case}
        if ($S_i \cap cur\_set \neq \varnothing$) **then**
            search_max($i+1$, *min*, *max*, *cur_set*);
        else for each $C \in S_i$ **and** C $\notin$ cur_set **do**
            $cur\_set := cur\_set \cup C;$
            search_lhs($i+1$, *min*, *max*, *cur_set*);
            $cur\_set := cur\_set - C;$
  end search_max

procedure add_and_remove_supersets(set,new_element)
{Remove supersets of new_element from set and add new_element to set.}

Algorithm 3.8. Find one Lossless Join Decomposition into BCNF [Tsou82]

Input: A set of attributes R, a set of functional dependencies F.

Output: A decomposition of R.

S is a subset of R.
S is a subset of S.
last is an attribute from T.

```
S:=R;
modified:=true;
while modified do
    T:=S;
    modified:=false;
    last:=∅;
    while |S|>2 do
        for each A∈T do
            for each B∈T−A do
                if T−AB→A∈F⁺ then
                    T:=T−B;
                    last:=A;
                    modified:=true;
    S:=S−last;
    decomp:=decomp∪T;
```

one more decomposition $\{AB,AC\}$. We have now found all the possible decompositions from this dependency set. Note that the decomposition $\{AB,AC,BC\}$ cannot be found by the Fisher-Tsou algorithm. □

This is a simple case, and we need to do some further manipulations to make sure that the relation really is in BCNF. The basis $B(F)$ may contain dependencies $XY \rightarrow AZ$ and $A \rightarrow Y$. If we want to remove A, we get the relation XYA, which is not in BCNF, because $A \rightarrow Y$ holds over it. This means that we must manipulate $B(F)$ so that we only generate relations in BCNF. That can be done by expanding $B(F)$ so that the dependencies only have one attribute on the right-hand side. Then remove all dependencies $X \rightarrow A$ where XA is a superset of the union of some other dependencies left- and right-hand sides. This will remove the possibility to generate relations that are not in BCNF.

Algorithm 3.9. Find All Possible BCNF Decompositions

Input: A set of attributes R, a set of functional dependencies F.
Output: All the BCNF decompositions.

1. {Find the basis $B(F)$}

2. {Remove $X \rightarrow A$ if $XA \subseteq YB$ where $Y \rightarrow B \in B(F)$ producing set D}

3. {Find all the BCNF decompositions}

```
    cur_decomp:=∅;
    all_decomps:=∅;
    find_all_bcnf(R,D,cur_decomp,all_decomps);


procedure find_all_bcnf(S, D, cur_decomp, all_decomps)
{add all decompositions to all_decomps}

    if |S|≤2 then
        all_decomps:=all_decomp∪{cur_decomp∪S};
    else
        for each A∈S do
            a_deps:={all dependencies containing A};
            for each X→A do
                find_all_bcnf(S−A, D−a_deps,
                              cur_decomp∪{XA}, all_decomps);


end find_all_bcnf
```

Algorithm 3.9 describes this method. Unfortunately the method is exponential. In addition, we cannot guarantee that we do not produce any duplicate candidate decompositions. Further research is needed on these issues. The Tsou-Fischer algorithm also has a serious drawback since it has a tendency to produce BCNF decompositions that are not dependency preserving, and it may not find a dependency preserving decomposition even if such a decomposition exists.

3.11. Summary

In this chapter, we presented two new graph representations of functional dependencies and described efficient design algorithms based on manipulating these graphs.

The semilattice representation led us to the definition of a basis $B(F)$, which can be used to generate covers (Section 3.3), construct a generator for the F-closed sets (Section 3.9), and find all Boyce-Codd normal form decompositions of a relational scheme (Section 3.10).

The attribute-fd graph gave us a unified way of representing functional dependencies that could be used by the most common design algorithms. The design algorithms for finding the closure, the projection of dependencies, and finding the keys could be viewed as traversing the graph; along the edges and/or against the edges, depending on how the transitivity and pseudotransitivity rule was applied to the functional dependencies (Section 3.5, Section 3.6, and Section 3.7). The graph was also used to compute the basis $B(F)$ (Section 3.7).

The use of the attribute-fd graph has resulted in design algorithms that have complexities equal to or less than previously presented algorithms. The closure algorithm (Section 3.5) is $O(|F|)$, the same as the Beeri and Bernstein algorithm [Beer79]. The advantage of our algorithm is its simplicity, its disadvantage is that the attribute-fd graph requires the dependencies to be in reduced form.

The projection algorithm presented (Section 3.6) has complexity $O(\|R\| \cdot |F|^{2\|R\|})$ compared to the $O(\|R\|^5 \cdot |F|^{3\|R\|})$ complexity of Gottlob's projection algorithm. We showed that for some dependency classes the complexity of finding the basis $B(F)$ (Section 3.7) will be polynomial. The key finding algorithm (Section 3.8) is polynomial in the number of keys found, the same as for the Luccehsi and Osborn algorithm but we give some heuristics to reduce the number of candidate keys produced.

Finding the generator (Section 3.9) has, in the worst case, exponential time complexity in the number of attributes. Mannila and Räihä [Mann85] showed that for a set of dependencies over a normalized scheme this is substantially less. Finding the various BCNF decompositions, which can be produced by the Fischer and Tsou polynomial algorithm, has also, in the worst case exponential time complexity.

CHAPTER 4

Multivalued Dependencies and Join Dependencies

Multivalued dependencies have been studied by researchers over a number of years. Despite the fact that multivalued dependencies are useful in database design, they are not used by database designers as frequently as functional and inclusion dependencies.

A multivalued dependency describes a many-to-many or one-to-many relationship. A multivalued dependency also indicates that an attribute set is independent of another attribute set. Because of this property, the multivalued dependencies provide guidelines on how to split attributes into multiple tables. Multivalued dependencies are called *structural* [Beer86a], since the dependency should be used to structure the database. Functional dependencies, on the other hand, are not always structural. Sometimes they describe integrity constraints that should be maintained within a relation table, thus they should not be used to split a database into multiple tables.

Join dependencies are a generalization of multivalued dependencies. In other words, a multivalued dependency is a special case of a join dependency. Join dependencies are important because they state that relations in a database scheme can be joined losslessly. A logical database design that has the lossless join property is considered a must, because this propery assures data consistency. Join dependencies, like multivalued dependencies, should be considered structural. As we shall see later, a join dependency is often equivalent to a set of multivalued dependencies.

In Section 4.1, we will describe multivalued dependencies and join dependencies, their properties, and other useful concepts for database designers. Section 4.2 contains a survey on previous research. In Section 4.3, we will present an algorithm to infer multivalued dependencies from a special class of join dependencies.

4.1. Concepts and Definitions

We say that the set of attributes X multidetermines the set of attributes Y, $X \rightarrow\rightarrow Y$, if each occurrence of an X-value is always associated with a specific set of Y-values, and that these Y-values are not logically related to values of any other attributes other than X. A functional dependency is a special case of a multivalued dependency, where each X-value is associated with only one Y-value. Formally the definition of multivalued dependency is as follows:

Definition 4.1: *Multivalued dependencies.* We say that relation **r** satisfies the multivalued dependency $X \rightarrow\rightarrow Y$ if for every pair of tuples t_1 and t_2 in **r**, such that $t_1[X]=t_2[X]$ there exists t_3 in **r** such that

1. $t_1[X]=t_2[X]=t_3[X]$
2. $t_1[Y]=t_3[Y]$
3. $t_2[\mathbf{R}-X-Y]=t_3[\mathbf{R}-X-Y]$

where $t[Z]$ stands for t restricted to domain Z, and R is the union of all attributes in **R**. Note that this definition is symmetric in Y and $R-Y$. Thus $X \rightarrow\rightarrow Y$ implies $X \rightarrow\rightarrow R-Y$, assuming that X and Y are disjoint.

Example 4.1: Consider a database with attributes Course, Teacher, Room, Hour, Student, and Grade. A course can be scheduled to meet several times a week and students attend all the meetings. This can be expressed by the multivalued dependency {Course, Teacher} $\rightarrow\rightarrow$ {Hour, Room}. An instance of the database table is given below.

Course	Teacher	Room	Hour	Student	Grade
CS482	Clark	Tue 9:00	U Hall 100	Peter	A
CS482	Clark	Tue 9:00	U Hall 100	Paul	B
CS482	Clark	Fri 10:00	L Hall 300	Peter	A
CS482	Clark	Fri 10:00	L Hall 300	Paul	B
CS421	Taylor	Wed 10:00	L Hall 100	Peter	B
CS421	Taylor	Wed 10:00	L Hall 100	Paul	A

Figure 4.1. University Database

Inspecting the table instance we see that there is no close relationship between attribute sets {Room, Hour} and {Student, Grade}. We say that these sets are *indirectly connected* via {Course, Teacher} [Beer86a]. We also see that every {Room, Hour}-value for a course is associated with a list of {Student, Grade}-values. The list of students attending a course is repeated every time it is scheduled to meet. This is not space-efficient, and the multivalued dependency {Course, Teacher} $\rightarrow\rightarrow$ {Room, Hour} suggests that {Room, Hour} and {Student, Grade} should be in different relations to avoid repetition. Using multivalued dependencies in database design will always remove redundancy, a major goal in database design, multivalued dependencies are *structural.* □

There are rules that can be used to derive multivalued dependencies implied by a given set of multivalued dependencies:

Complementation.
If $X \rightarrow\rightarrow Y$ is a multivalued dependency over attributes R, then $X \rightarrow\rightarrow R - Y$.

Reflexivity.
If $Y \subseteq X$, then $X \rightarrow\rightarrow Y$.

Augmentation.
If $Z \subseteq W$ and $X \rightarrow\rightarrow Y$, then $XW \rightarrow\rightarrow YZ$.

Transitivity.
If $X \rightarrow\rightarrow Y$ and $Y \rightarrow\rightarrow Z$, then $X \rightarrow\rightarrow Z - Y$.

Pseudotransitivity.
If $X \rightarrow\rightarrow Y$ and $YW \rightarrow\rightarrow Z$, then $XW \rightarrow\rightarrow Z - YW$.

Union.
If $X \rightarrow\rightarrow Y$ and $X \rightarrow\rightarrow Z$, then $X \rightarrow\rightarrow YZ$.

Decomposition.
If $X \rightarrow\rightarrow Y$ and $X \rightarrow\rightarrow Z$, then $X \rightarrow\rightarrow Y \cap Z$, $X \rightarrow\rightarrow Y - Z$, and $X \rightarrow\rightarrow Z - Y$.

The three first multivalued dependency axioms are sufficient, and the remaining axioms are implied by them [Ullm82]. In addition, we have inference rules that use both functional and multivalued dependencies. They are:

Generalization from Functional to Multivalued Dependency.
If $X \rightarrow Y$, then $X \rightarrow\rightarrow Y$.

Mixed Pseudotransitivity.
If $X \rightarrow\rightarrow Y$ and $XY \rightarrow Z$, then $X \rightarrow Z - Y$.

The inference rules can be used to decide whether a specific dependency can be derived from a given dependency set. For functional dependencies, this is done by computing the closure of a set of attributes. For multivalued dependencies, this is done by finding the so called *dependency basis* of a set of attributes, defined as follows:

Definition 4.2: *Dependency Basis.* The *dependency basis* of a set of attributes $X=A_1A_2..A_n$, denoted $DEP(X)$ is a set of minimal sets $\{A_1,A_2,...,A_n,Y_1,Y_2,...,Y_m\}$ such that $X \rightarrow\rightarrow Y_i$ and all Y_i's are disjoint.

The multivalued dependency $X \rightarrow\rightarrow Z$ holds if and only if Z is a union of one or more of the elements of $DEP(X)$.

Example 4.2: The dependency basis of {Course, Teacher} of Example 4.2 is {{Course}, {Teacher}, {Hour, Room}, {Student, Grade}}. □

Recall that given a set of functional dependencies F over R, in order to check whether the functional dependency $X \rightarrow Y$ holds for X, $Y \subseteq R$, it is sufficient to test whether $Y \subseteq X^+$, where X is the closure of X with respect to F. The test of implication for a set of functional and multivalued dependencies is done by using (i) the dependency basis for multivalued dependencies, (ii) the closure for functional dependencies, and (iii) the axiom for mixed pseudotransitivity [Beer80].

Two multivalued dependency sets G_1 and G_2 are *equivalent* if they produce the same dependency basis for all attribute sets. We can also say that for equivalent multivalued dependency sets G_1 and G_2, G_1 *covers* G_2 and G_2 *covers* G_1. Two multivalued dependency sets are equivalent if the dependency basis for X taken with respect to G_1 is identical to the dependency basis taken with respect to G_2 where $X \rightarrow\rightarrow Y$ is either in G_1 or G_2.

A multivalued dependency $X \rightarrow\rightarrow Y$ is *redundant* if the dependency basis of X remains unchanged after removing $X \rightarrow\rightarrow Y$ from dependency set G [Beer80]. This means that $G - \{X \rightarrow\rightarrow Y\}$ is a cover for G.

The relationship between multivalued dependencies and lossless joins is described in the following theorem.

Theorem 4.1: [Maie83a] Relation instance **r** satisfies the multivalued dependency $X \rightarrow\rightarrow Y$, where X, $Y \subseteq R$, if and only if **r** decomposes losslessly onto the relations $R_1 = X \cup Y$ and $R_2 = (R - Y) \cup X$.

Decomposing the relations according to this theorem results in normalizing into fourth normal form (4NF). Fourth normal form is defined as:

Definition 4.3: *Fourth Normal Form.* Let F be a set of functional and multivalued dependencies that hold for a relation R. R is in fourth normal form if every multivalued dependency $X \rightarrow\rightarrow Y$ implied by F either are trivial or X is a superkey for R (i.e, $X \rightarrow R$).

Example 4.3: The multivalued dependency in Example 4.2 would produce the 4NF relations {Course, Teacher, Room, Hour} and {Course, Teacher, Student, Grade}. □

For most real-world applications, Sciore [Scio81] claims that the multivalued dependencies will be *conflict-free*, where conflict-free is defined as follows:

Definition 4.4: *Conflict-Free.* A set of multivalued dependencies is considered to be conflict-free if the two following properties hold.

(1) No *split left-hand side anomaly* occurs. This means that no left-hand side of a multivalued dependency intersects with more than one element of a dependency basis of another left-hand side.

(2) No *intersection anomaly* occurs. This means that for any two left-hand sides X and Y, the property $DEP(X \cap Y) \supseteq DEP(X) \cap DEP(Y)$ holds.

Example 4.4: Consider again the database of Example 4.2 with attributes {Course, Teacher, Hour, Room, Student, Grade}. Let us add the attribute Major to this database. The university has the requirement that all students must take the courses in their major. This means that the students have no liberty to choose their own courses, but must follow a preassigned program by the university. A student can only have one major. This gives the following multivalued dependency:

(1) {Major}$\rightarrow\rightarrow$ {Course} | {Student} | {Teacher, Hour, Room, Grade}
DEP({Major})={{Major}, {Course}, {Student}, {Teacher, Hour, Room, Grade}}

In addition, the multivalued dependency from Example 4.2 now becomes:

(2) {Course, Teacher}$\rightarrow\rightarrow${Hour, Room} | {Student, Grade} | {Major}
DEP({Course, Teacher})={{{Course}, {Teacher}, {Hour, Room}, {Student, Grade}}

we see that the left-hand side of multivalued dependency 2 is split by multivalued dependency 1 since the attributes Course and Teacher are contained in two different elements of the dependency basis of Major. □

Multivalued dependency sets with the split left-hand side anomaly are undesirable because we can get into a conflict when we try to decompose the relation into 4NF. In the above example, we get the relations {Major, Course} and {Major, Student, Teacher, Hour, Room, Grade} when we decompose with respect to the multivalued dependency {Major}$\rightarrow\rightarrow$ {Course}. If we try to apply the multivalued dependency {Course, Teacher}$\rightarrow\rightarrow${Hour, Room} to the resulting relations, we see that we are no longer able to decompose with respect to this dependency since none of the relations contain both Course and Teacher. For a dependency set without the split left-hand side, such a situation will not arise.

Example 4.5: Consider the dictionary database [Scio82] of technical terms in English, German, and French. Each word can stand for only one concept, but there can be several words in any language that stand for one word in any other language. We can describe this with the multivalued dependencies {English}$\rightarrow\rightarrow${German} | {French}, {German}$\rightarrow\rightarrow$ {English} | {French} , {French}$\rightarrow\rightarrow$ {English} | {German}. The dependency basis for the attributes are identical; *DEP*({English})= *DEP*({German})= *DEP*({French})= {{English},{German},{French}}. We see that *DEP*({English}$\cap${German})={$\varnothing$,{English, German, French}} and that *DEP*({English})$\cap$*DEP*({German})= {{French}}. The dependency set has an intersection anomaly since {{French}}$\not\subseteq${$\varnothing$,{English, German, French}}. Inspecting the dependency basis of the different attributes suggest that the words in the different languages are not related to each other. The multivalued dependency $\varnothing\rightarrow\rightarrow${English} | {German} | {French} describes this fact better. But in practice this makes no sense. □

Sciore extensively discusses the split left-hand side and intersection anomalies [Scio82] and concludes that these types of multivalued dependencies are unnatural. He claims that when a set of multivalued dependencies has the split left-hand side anomaly and/or the intersection anomaly, the dependency set is erroneous and should be rewritten to contain none of these anomalies. This will be discussed further in Chapter 5.

According to Theorem 4.1 the multivalued dependencies can be used to detect lossless decompositions. The join dependency (JD) also describes a lossless decomposition. Join dependencies are formally defined as follows:

Definition 4.5: *Join Dependency.* A relation **r** over **R** satisfies the join dependency $*(R_1,R_2,\ldots,R_m)$ if $\mathbf{r} = \pi_{R_1}(\mathbf{r}) * \ldots * \pi_{R_m}(\mathbf{r})$; that is, when we project **r** onto the sets of attributes R_i and then join the resulting relations, we get back the relation instance **r** and no extraneous tuples. We can also say that $R_1 * R_2 * \ldots * R_n$ is lossless. Note that the multivalued dependency $X \rightarrow\rightarrow Y$ can be defined as the join dependency $*(XY,R-Y)$.

We say that a join dependency $*(R_1,R_2,\ldots,R_m)$ is trivial if some R_i=**R**. The join dependencies also have inference rules. We list the most common among these rules below:

Universal Axiom

$*\mathbf{R}$ is always true.

Covering

$*\mathbf{S}=*(S_1,S_2,\ldots,S_m) \Rightarrow *\mathbf{R}*(R_1,R_2,\ldots,R_m)$ if $S_i \subseteq R_i$ foe all $i \in [1,m]$.

Generalization from Functional to Join Dependency

$X \rightarrow Y \Rightarrow *(XY,(R-Y) \cup X)$

Generalization from Multivalued to Join Dependency

$X \rightarrow\rightarrow Y \Rightarrow *(XY,(R-Y) \cup X)$

Translation from Join Dependency to Multivalued Dependency

$*(X,Y) \Rightarrow X \cap Y \rightarrow\rightarrow Y-X | X-Y$

A relation scheme $\mathbf{R}=\{R_1,R_2,\ldots,R_p\}$ or join dependency $*(R_1,R_2,\ldots,R_p)$ can be represented by a hypergraph, where a hypergraph is defined as follows:

Definition 4.6: *Hypergraph.* The pair H=(**V**,**E**) is called a hypergraph, where $\mathbf{V} = \{A_1,A_2,\ldots,A_n\}$ is a set of vertices and $\mathbf{E} = (R_1,R_2,\ldots,R_m)$ is a set of hyperedges, that is, each R_i is a subset of **V**. An ordinary graph is a hypergraph where all edges have cardinality of two.

Example 4.6: Consider six attributes A, B, C, D, E, F and the relation scheme $\mathbf{R}=\{ABC, CDE, AEF, ACE\}=\{R_1,R_2,R_3,R_4\}$. R is represented by the hypergraph $H=(\{A, B, C, D, E, F\},\mathbf{R})$ in Figure 4.2. □

The hypergraphs we are working with will be *reduced* hypergraphs, where no hyperedge is a subset of another hyperedge. Before we define cyclicity and acyclicity of hypergraphs we need to define the concept of induced subhypergraph, articulation set, and block of a hypergraph.

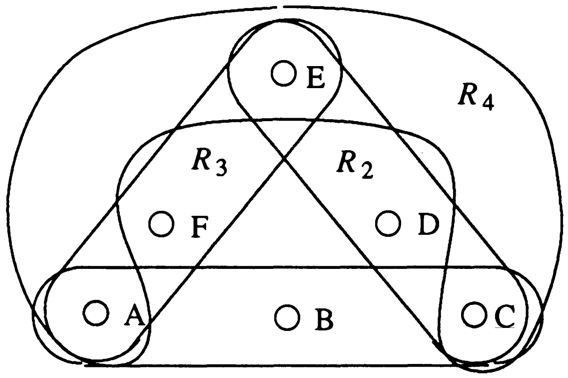


Figure 4.2. Hypergraph Representation of R

Definition 4.7: *Induced Subhypergraph.* Let H=(**V**,**E**) be a hypergraph and let **M** be a subset of **V**. The subhypergraph H' induced by the vertices **M**, is the reduction of hypergraph $H'=(\mathbf{M},\{E_i \cap \mathbf{M} \mid E_i \in \mathbf{E}\})$.

Example 4.7: Consider the hypergraph in Figure 4.2. The hypergraph induced by the vertices $\{A, C, E\}$ is the the single hyperedge R_4. □

Definition 4.8: *Articulation Set.* Let A and B be hyperedges in the reduced hypergraph H. Then $Q=A \cap B$ is an *articulation set* if and only if removing the vertices in Q from every hyperedge in H strictly increases the number of connected components of H.

Example 4.8: The hypergraph in Figure 4.2 has three articulation sets {A, E}, {A, C}, and {C, E}. The set {A, C, E} is not an articulation set because it is not an intersection of two hyperedges. □

Definition 4.9: *Block of a Hypergraph.* A block is a connected induced subhypergraph with no articulation sets. A block is trivial if it consists of a single hyperedge or a subset of a hyperedge. A hypergraph may contain many blocks.

Example 4.9: The only blocks of the hypergraph in Figure 4.2 are subsets of R_4. As soon as one of the vertices F, B, or D is included, the induced subhypergraph will not be a block since it has an articulation set. □

Definition 4.10: *Knob.* A vertex that is contained in only one hyperedge is called a *knob.*

Definition 4.11: *Acyclic and Cyclic Hypergraphs.* A reduced hypergraph is acyclic if all its blocks are trivial, otherwise it is cyclic.

Example 4.10: The hypergraph in Figure 4.2 is acyclic, since as we noted that all blocks are subsets of hyperedge R_4, and thus every block is trivial. The hypergraph consisting of the hyperedges R_1, R_2, and R_3 in Figure 4.2 is a cyclic hypergraph. □

The fact that a hypergraph is acyclic does not mean that the hypergraph does not contain any cycles, as can be seen in the acyclic hypergraph in Figure 4.2. As the previous example shows, a cyclic hypergraph can be contained in an acyclic graph. The cycles in an acyclic hypergraph are not chordless, meaning that every cycle has a hyperedge that contains the intersection of at least three pairs of hyperedges. In our example the hyperedge that fulfills this property is hyperedge R_4. More formally, this is described by the following definition and theorem.

Definition 4.12: *Chordless* [Maie83a]. Let $H=(N,E)$ be a hypergraph and let $P=E_1,E_2,...,E_m$ be a path in H. Define $F_i=E_i \cap E_{i+1}$, $1 \leq i$. P is *chordless*, if there is no edge in E that contains $F_i \cup F_j \cup F_k$ $1 \leq i<j<k<m$. That is, no edge in E contains three or more intersections of adjacent edges in the path. P is a *cycle* if $E_1=E_m$.

Theorem 4.2: [Maie83a]. Hypergraph H is acyclic if and only if H contains no chordless cycles of three or more edges.

Testing whether a hypergraph is acyclic can seem to be inherently complex, since all induced subhypergraphs must be checked. Note that in the case where we added a knob to a block from the above example, the subhypergraph is no longer a block. This means while testing whether a hypergraph is acyclic, we need not consider subhypergraphs containing knobs. Thus, knobs can be removed, and the remaining reduced hypergraph will contain the same blocks as the original hypergraph, except for trivial blocks. Using this observation there is a simple algorithm called Graham's algorithm, which can determine whether a hypergraph is acyclic:
Graham's Algorithm [Grah79].
Apply the following three operations to $\mathbf{R}=\{R_1,R_2,...,R_p\}$ repeatedly until neither can be applied.

1. If A is an attribute that appears in exactly one R_i, then delete A from R_i. That is, remove all knobs.
2. Delete R_i if there is an R_j, with $j \neq i$, such that $R_i \subseteq R_j$. That is, reduce the remaining hypergraph.
3. Delete R_i if R_i contains no vertices.

When the algorithm is given a relation with an acyclic hypergraph, it will return the empty set. If it is a cyclic hypergraph, there will be two or more edges left. Graham's algorithm runs in time quadratic in the size of the hypergraph. There is also a linear-time algorithm for testing acyclicity by Tarjan and Yannakakis [Tarj84], using the maximum cardinality search for hypergraphs. This method will be described in Section 4.3.3.

Beeri et al. [Beer83] studied the relationship between join dependencies, multivalued dependencies, and acyclic relation schemes to prove the following theorem:

Theorem 4.3: [Beer83] The following conditions on R are equivalent:

1. $\mathbf{R}=\{R_1,R_2,...,R_p\}$ is an acyclic hypergraph.
2. $*\mathbf{R}=*(R_1,R_2,...,R_p)$ is equivalent to a conflict-free set of multivalued dependencies.

This theorem and the research done be Sciore, suggests that a database designer is better off designing relational databases where the multivalued dependencies are equivalent to one acyclic join dependency. This topic will be studied further in Chapter 4.

4.2. Survey of Previous Research

Multivalued dependencies were first introduced by Fagin [Fagi77a] together with the fourth normal form. The concept of multivalued dependencies was independently defined by Zaniolo [Zani76]. Delobel and Leonard [Delo74] previously defined a related concept.

Beeri, Fagin, and Howard [Beer77] described the first complete axiomatization for functional dependencies and multivalued dependencies. The notion of the dependency basis was given by Beeri [Beer80]. An algorithm to find the dependency basis, of time complexity $O(|D|^4)$ where D is the size of the dependency set, was also presented by Beeri. The algorithm could handle both functional dependencies and multivalued dependencies. More efficient algorithms to calculate the dependency basis have later been found by Hagihara, Ito, et al. [Hagi79], Sagiv [Sagi80], and Galil [Gali82]. The time complexity of the algorithms were $O(|D|^2)$, $O(|D|\cdot|R|)$, and $O(|D|\cdot(1+min(k+log\ p)$ respectively, where $|R|$ is the number of attributes in the database, k is the number of multivalued dependencies in D and p is the size of the dependency basis.

Fourth normal form is the most common normal form used for multivalued dependencies. A synthesis algorithm to normalize into fourth normal form is given by Beeri [Beer79]. Other normal forms and decomposition strategies are treated by many researchers [Beer86a, Maie83a].

Multivalued dependencies that hold only over a subset of R, called embedded multivalued dependencies were introduced by Fagin [Fagi77a] and Delobel [Delo78]. The notion of a conflict-free set of multivalued dependencies was given by Lien [Lien82]. A conflict-free set of multivalued dependencies is a set that has neither the split left-hand side anomaly nor the intersection anomaly. Sciore continued studying conflict-free multivalued dependencies and gave methods to eliminate these anomalies [Scio81]. Both Lien and Sciore claim that sets of multivalued dependencies that describe real-world applications are conflict-free. Beeri and Kifer [Beer86b] give an algorithm to detect intersection anomalies. The intersection anomalies are considered to be undesirable as they can result in redundancy problems for fourth normal form schemes [Fagi86b].

Join dependencies were first introduced in full generality by Rissanen [Riss77]. Aho, Beeri, and Ullman [Aho79a] studied join dependencies further in relationship with queries to relational databases. Sciore argues that join dependencies do not have a complete set of axioms [Scio82] and gives a complete set of axioms for a class of dependencies larger than the join dependencies, called full join dependencies. Beeri and Vardi [Beer81b] and then Vardi [Vard83] give inference axioms for functional and join dependencies. Vardi [Vard83] also gives inference axioms for multivalued and join dependencies and presents an $O(|D|\cdot|R|)$ algorithm to test whether a multivalued dependency is implied by a set of functional and join dependencies.

The definition of acyclic database scheme comes from Fagin, Mendelzon, and Ullman [Fagi82c]. Beeri et al. state that an acyclic join dependency is equivalent to a conflict-free set of multivalued dependencies [Beer83].

The methods known so far to infer multivalued dependencies from a join dependency test whether a multivalued dependency is inferred from a join dependency or from a set of functional and join dependencies [Vard83]. We extend this result in the following way. Given an acyclic join dependency, we show how to find a set of multivalued dependencies that are equivalent to the join dependency. Our main results are stated in Theorem 4.10 and 4.12 below.

4.3. Inferring Multivalued Dependencies from Join Dependencies

During logical design of a relational database, it is important that the designer can understand the semantic implications of a candidate design. A candidate scheme can be proposed by the designer, or generated by decomposition or synthesis based on a given set of dependencies. In chapters 3 and 6, we saw how dependencies can be inferred from a relation instance. In this section, we address the problem of inferring multivalued dependencies from a given scheme. There is a significant difference between inferring semantic information from a relation instance and from a scheme. The former extracts relationships that exist between attributes within a relation, the latter extracts relationships between the different relations in a scheme. Our assumption is that a database designer will tend to group attributes that belong together in one relation, and that attributes in different relations are independent of each other unless they appear in several relations. We also assume that the scheme $R_1,R_2,...,R_p$ suggested by the designer is lossless, meaning that we assume the join dependency $*(R_1,R_2,...,R_p)$. In fact, the notion of independence between attributes in different relations is the relationship that multivalued dependencies describe. Our problem is then to infer multivalued dependencies from one join dependency.

To find the multivalued dependencies, we represent the relational scheme as a graph. In the following section we will describe different graph representations for a relational scheme. We will then describe what multivalued dependencies are with respect to the graphs, and then give an algorithm to find the multivalued dependencies in the case where the relation scheme is acyclic. We will also discuss the case where the relation scheme is cyclic.

4.3.1. Graph Representation of Relational Schemes

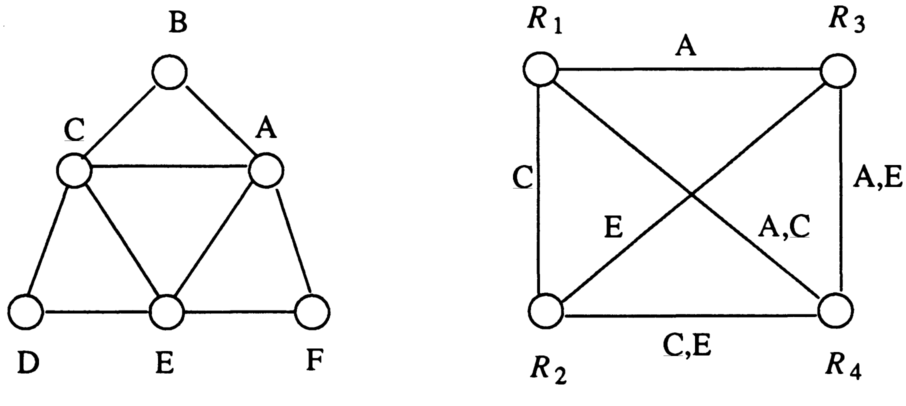
Any relational database scheme $R=\{R_1,R_2,...,R_p\}$ can be represented by a hypergraph. The vertices of the hypergraph will correspond to the attributes mentioned in the relations R_i, and the hyperedges correspond to the relations $\{R_1,R_2,...,R_p\}$ of the scheme. Thus, when we talk about vertices and hyperedges of hypergraphs one can think of attributes and relations of a scheme.

There are also ordinary graph representations for relation schemes. These representations can be thought of as a transformation from the hypergraph to the ordinary graph. The two most common representations are the ordinary graph representations G and the join graph representation J. These two graphs can be constructed from the hypergraph H as follows:

Definition 4.13: *Ordinary Undirected Graph G* [Beer83]. Each vertex of graph $G(H)$ corresponds to a vertex of H, and there is an edge between every pair of vertices that are in the same hyperedge of H. See Figure 4.3 a.

Definition 4.14: *Join Graph J* [Maie83a]. Each vertex in J(H) corresponds to a hyperedge in H. There is an edge (h_i,h_j), if hyperedge h_i intersects with hyperedge h_j in H. In order to get a one-to-one correspondence, we need to label the edges with $h_i \cap h_j$. See Figure 4.3 b.

There are advantages and disadvantages to both of the above graph representations. The ordinary graph $G(H)$ can be constructed in a simple manner by making every hyperedge into a clique. The drawback of this representation is that there is not always a one-to-one mapping between $G(H)$ and H, as the following example shows.



a. Ordinary Graph

b. Join graph

Figure 4.3. Graph Representations of R

Example 4.11: Consider the subhypergraph H', consisting of the hyperedges R_1, R_2, R_3 of the hypergraph H in Figure 4.2. The ordinary graph representation $G(H')$ is equivalent to $G(H)$ in Figure 4.3 a. Thus, we do not know whether hypergraph H or H' produced the ordinary graph. □

However, if we restrict H to be acyclic, then there is a one-to-one mapping between H and $G(H)$ [Beer83].

For the join graph J(H), on the other hand, there is a one-to-one mapping, provided that the edges are labeled. If p is the number of relations, the time to construct the join graph is $O(2^p)$, since we must determine the intersection between all hyperedges of H in order to find the labels of the edges. In general, this is greater than constructing the ordinary graph, which is $O(p \cdot 2^k)$ where k is the maximal size of a hyperedge and p is the number of hyperedges.

In our algorithm to infer multivalued dependencies from a join dependency, we will use the hypergraph representation and the ordinary graph representation.

4.3.2. Multivalued Dependencies in Graph Terminology

As mentioned in Section 4.1, multivalued dependencies are a special case of join dependencies. Instead of saying that the join dependency $*(R_1, R_2)$ holds, we can say that we have the multivalued dependency $R_1 \cap R_2 \rightarrow\rightarrow R_1 - R_2 | R_2 - R_1$. Thus, if $R = \{R_1, R_2\}$ is given and assumed lossless, we can represent scheme R by a hypergraph H. From H we can generate the multivalued dependency $R_1 \cap R_2 \rightarrow\rightarrow R_1 - R_2$ since the removal of $R_1 \cap R_2$ separates or divides the hypergraph H into two components, $R_1 - R_2$ and $R_2 - R_1$.

More generally, given any hypergraph H the set of multivalued dependencies generated by H is the set of multivalued dependencies $X \rightarrow\rightarrow Y$, where X and Y are disjoint, and Y is the union of some connected components of the hypergraph $H - X$ obtained by removing X from H [Beer83]. We say that X is a *separator* of the hypergraph H. A separator is defined as follows:

Definition 4.15: *Separator of a Hypergraph.* A set of vertices X is a *separator* of the hypergraph H, if the removal of X disconnects the hypergraph. X is a minimal separator if for all vertices $A \in X$, where $X - A$ is not a separator.

Example 4.12: Consider the hypergraph H in Figure 4.2. $S_1 = \{A, C\}$ and $S_2 = \{A, C, E\}$ are separators. S_1 will disconnect H into the connected components $\{B\}$ and $\{D, E, F\}$. This means that H generates the multivalued dependencies $\{A, C\} \rightarrow\rightarrow \{B\}$ and $\{A, C\} \rightarrow\rightarrow \{D, E, F\}$. S_1 is a minimal separator but S_2 is not. □

Finding a set of implied multivalued dependencies of a relation scheme is equivalent to finding all the separators in its corresponding hypergraph. In general, this task can be of complexity $O(2^n)$. For the time being, we will concentrate on the case where the relation scheme is acyclic. For this case, the number of multivalued dependencies is limited to $p-1$, where p is the number of relations in the scheme [Beer83]. It has also been shown that this set of multivalued dependencies is conflict-free.

In the remaining portion of this section we will show that in the case that R is acyclic, we can infer the multivalued dependencies by finding a subclass of separators, the articulation sets. Articulation sets are separators, but not all separators are articulation sets. The hypergraph in Figure 4.2 has three articulation sets, {A, E}, {A, C}, and {C, E}. These articulation sets are all separators, but the separator {A, C, E} is not an articulation set.

In general, there is no relationship between minimal separators and articulation sets. A cyclic scheme may not have any articulation sets but may still have minimal separators.

Example 4.13: Consider again the subgraph H' consisting of the hyperedges R_1, R_2, R_3 of the hypergraph in Figure 4.2, H' has no articulation sets but it has the minimal separators, {A,E}, {A,C}, and {C,E}. □

We conjecture that for, acyclic schemes, the minimal separators of the hypergraph representing the scheme are the articulation sets of the hypergraph.

Since we will use both the hypergraph representation and the ordinary graph representation, we need to prove the following lemma.

Lemma 4.4: Every separator of the graph $G(H)$ is a separator of the reduced hypergraph H.

Proof: Let S be a separator of $G(H)$ and say that $G(H)-S$ contains two components C_A and C_B where $A \in C_A$ and $B \in C_B$. We must now show that $H-S$ also contains two different components one containing A, the other B. Assume there are no such components. This means that there is a path $R_1, R_2, ..., R_k$ in $H-S$ such that $A \in R_1$ and $B \in R_k$. A path in $H-S$ means that $R_i \cap R_{i=1} \neq \emptyset$ for all i. But then $G(H-S)$ must also contain a path from A to B. But clearly $G(H)-S$ must be identical to $G(H-S)$ since both graphs use the same vertices and the edges of $G(H)-S$ can be expressed by $\{e_i \mid |e_i|=2$ and $\forall\ R_j\ e_i \in 2^{R_j}\} - \{e_i \mid |e_i|=2$ and $\forall\ R_j\ e_i \in 2^{R_j}$ and $e_i \cap S \neq \emptyset\}$ that is identical to $\{e_i \mid |e_i|=2$ and $\forall\ R_j\ e_i \in 2^{R_j} - S\}$, which are the edges of $G(H-S)$. This contradicts that $G(H-S)$ contains a path between A and B, which contradicts that $H-S$ contained a path between A and B. □

Now that we have showed that we can use the ordinary graph to find separators for the hypergraph, we also need the following lemma.

Lemma 4.5: Let $R=\{R_1, R_2, \ldots, R_p\}$ be an acyclic scheme. Then every separator of hypergraph H representing R is contained in one hyperedge.

Proof: According to Beeri et al. [Beer83] if H is acyclic then $G(H)$ is chordal and the maximal cliques in $G(H)$ correspond precisely to the hyperedge of H. An ordinary graph is chordal if every cycle with four or more edges has a chord, which is an edge joining two vertices that are not adjacent on the cycle. A clique S is a set of vertices that are pairwise adjacent. The clique S is maximal if no other vertex of the graph is adjacent to all the vertices in S. Golumbic [Golu80] shows that every minimal separator in a chordal graph is a clique. Because every maximal clique of $G(H)$ corresponds to a hyperedge, we can conclude that any separator of $G(H)$ being a clique, must be contained in a hyperedge of H. □

The running intersection property was defined by Beeri et al. because it is a convenient tool for proving properties of acyclic hypergraphs. The ordering also gives a monotone join expression. Given a join involving three or more relations $R_1,R_2,\ldots,R_p$, can be represented as a join expression $(((R_1*R_2)*R_3)*\cdots*R_p))$. Monotone join expressions guarantee that no intermediate two-way join has more tuples than the final join. The monotone join expressions gives a space-efficient manner for doing a joins.

Definition 4.16: *Running Intersection Property.* Let $R_1,R_2,...,R_p$ be a database scheme. The scheme has the *running intersection property* if there is a permutation $S_1,S_2,...,S_m$ of $R_1,R_2,...,R_m$ such that $(S_1,S_2,...,S_{i-1}) \cap S_i \subseteq S_j$ where $1<i\leq p$ and $j<i$.

Beeri et al. showed the following theorem:

Theorem 4.6: [Beer83] The following conditions on R are equivalent:

1. R is an acyclic hypergraph.
2. R has the running intersection property.

Example 4.14: An acyclic relation scheme can have several orderings that satisfy the running intersection property. Consider the relation scheme represented by the hypergraph in Figure 4.2. Inspecting the graph we find that both $\{R_4,R_1,R_2,R_3\}$ and $\{R_1,R_4,R_2,R_3\}$ satisfy the running intersection property, and there are clearly many more such orderings. □

In the following lemmas we will use the running intersection property to find articulation sets in acyclic relational schemes.

Theorem 4.7: If $R=\{R_1,R_2,\ldots,R_p\}$ has the running intersection property, then the sets $R_i \cap (R_1 \cup,...,\cup R_{i-1})$ for all i are articulation sets of the hypergraph representing R.

Proof: We must show that $R_i \cap (R_1 \cup,...,\cup R_{i-1})$ for any i, separates the hypergraph representing R. Let us select any i, $1 \leq i \leq$p. We know that $R_i \cap (R_1 \cup,...,\cup R_{i-1}) \subseteq R_j$ where $j<i$. Clearly, $R_i \cap (R_1 \cup,...,\cup R_{i-1}) = R_i \cap R_j$ since $j \in [1,i\text{-}1]$. We shall show that $R_i \cap R_j$ separates the hypergraph by showing that any hyperedge k numbered higher than i has the property that $R_k \cap (R_1 \cup,...,\cup R_{i-1}) \subseteq R_i \cap R_j$. Since R satisfies the running

intersection property we can state that $R_k \cap (R_1 \cup,...,\cup R_{k-1}) \subseteq R_l$. We now have the following cases. (i) $l>i$. In this case we can continue our investigation with the hyperedge numbered l. (ii) $l=i$. In this case $R_i \cap R_j$ will be an articulation set since $R_k \cap R_t \subseteq R_i$ for $t<i$ and $R_i \cap R_t \subseteq R_j$ for $t<i$ we get that $R_i \cap R_t \subseteq R_i \cap R_t \subseteq R_i \cap R_j$. (iii) For $l<i$, we must show that $R_k \cap R_l \subseteq R_i \cap R_j$. From the running intersection property we get:

1. $R_i \cap R_l \subseteq R_i \cap R_j$ since $l<i$, and
2. $R_k \cap R_i \subseteq R_k \cap R_l$.

From these two statements we get: $(R_i \cap R_l) \cap (R_k \cap R_i) = R_k \cap R_l \cap R_i \subseteq R_j \cap (R_k \cap R_l \cap R_i)$. From this we get $R_k \cap R_l \cap R_i \subseteq R_j$. But then $R_k \cap R_l \cap R_i \subseteq R_i \cap R_j$ is also true. Using the second statement above, we find that $R_k \cap R_l \subseteq R_i \cap R_j$. This shows that the removal of $R_i \cap R_j$ will disconnect the hypergraph into a connected component containing $R_1, \ldots, R_{i-1}$, and components containing $R_i, \ldots, R_p$. We have now showed that $R_i \cap (R_1 \cup,...,\cup R_{i-1}) = R_i \cap R_j$ for some $j<i$ is an articulation set for all $i \in [2,p]$. □

Thus, if we have an ordering that satisfies the running intersection property, we will be able to find all the articulation sets of the hypergraph H representing the scheme by finding $R_i \cap (R_1 \cup,...,\cup R_{i-1})$ for all i.

4.3.3. Finding All Articulation Sets of an Acyclic Hypergraph

Now that we have showed the correspondence between articulation sets and multivalued dependencies for acyclic schemes we need to develop an efficient algorithm that can find all the articulation sets. First, we produce an ordering of the relations that satisfies the running intersection property.

We know that for ordinary graphs the process of finding biconnected components, or all the articulation points, is linear in the number of edges. We would like to develop an algorithm for finding articulation sets of an acyclic hypergraph that is linear in the size of the number of vertices used to describe the hypergraph.

Tarjan and Yannakakis [Tarj84] give a linear-time algorithm to recognize acyclic hypergraphs. They use a *maximal cardinality search* on hypergraphs, defined as follows:

Definition 4.17: *Maximal Cardinality Search on Hypergraphs* [Tarj84]. Number the vertices n to 1 in decreasing order. As the next vertex to number, select any unnumbered vertex in an hyperedge containing as many numbered vertices as possible.

The maximal cardinality search for hypergraphs can be considered as finding the selection ordering $\{R_1, R_2, ..., R_p\}$ of the hyperedges, where all the vertices of R_i have been numbered before all vertices of R_j with $j>i$. The index of the lowest numbered hyperedge containing the vertex v is denoted as $\beta(v)$.

Tarjan and Yannakakis [Tarj84] showed the following theorem relating the ordering found by the maximal cardinality search and acyclicity of the hypergraph.

Theorem 4.8: The hypergraph is acyclic if and only if for all R_i from the maximal cardinality search ordering $\{R_1, R_2, ..., R_p\}$ of hyperedges satisfies the following condition:

$$R_i \cap \{ v \mid v \in R_k, 1 <= k \leq j \} \subseteq R_j \text{ where } j = max(\beta(v) \mid v \in R_i \text{ and } \beta(v) < i).$$

By inspecting this property one can see that it is a variation of the running intersection property. Tarjan and Yannakakis have thus specified an ordering that satisfies the running intersection property, without explicitly stating this result in their paper. We can also see that the maximal cardinality search can be used to find the hyperedge R_j that will contain the intersection of R_i with all the hyperedges numbered less than R_i. That is, for the ordering $R_1, R_2, ..., R_p$ of hyperedges found by the maximal cardinality search on hypergraphs, the following property holds:

$$(R_1 \cup ... \cup R_{i-1}) \cap R_i \subseteq R_j \text{ where } j = max(\beta(v) \mid v \in R_i \text{ and } \beta(v) < i).$$

We can now state the following lemma:

Lemma 4.9: The maximal cardinality search on acyclic relation schemes produce an ordering $R_1, R_2, ..., R_p$ that satisfies the running intersection property. Since the search takes time $O(n + \sum_{i=1}^{p} |R_i|)$ [Tarj84], this is also the complexity of finding the desired ordering, where n is the number of attributes.

We see that the articulation sets of an acyclic hypergraph consist of all the vertices in the next hyperedge to be numbered during the maximal cardinality search. Thus the following theorem can be stated:

Theorem 4.10: An acyclic hypergraph, with the hyperedges $R_1, R_2, ..., R_p$, has $p-1$ articulation sets. These sets can be found in time $O(n + \sum_{i=1}^{p} |R_i|)$.

Proof: According to Theorem 4.7, the articulation sets are intersections defining the running intersection property. The ordering of $R_1, R_2, ..., R_p$, which satisfy the running intersection property, can be found in time $(n + \sum_{i=1}^{p} |R_i|)$ using the maximal cardinality search. A linear search through all the p hyperedges for vertices where $\beta(v) < i$ takes time $O(n + \sum_{i=1}^{p} |R_i|)$ and will produce the $p-1$ articulation sets. □

Thus, we have an efficient algorithm to find articulation sets in acyclic hypergraphs. In the next section, we will describe how we can find the multivalued dependencies using the articulation sets.

4.3.4. Finding the Multivalued Dependencies

We pointed out earlier that a separator of a hypergraph corresponds to the left-hand side of a multivalued dependency for a relation scheme represented by the hypergraph. For the acyclic case, these separators reduce to articulation sets. Fagin et al. showed the following theorem:

Theorem 4.11: [Fagi82c] The set of multivalued dependencies implied by a join dependency $*(R_1,R_2,...,R_p)$ is exactly the set of multivalued dependencies generated by the hypergraph R.

We want to show that in the acyclic case, all the multivalued dependencies generated by the hypergraph R are implied by the multivalued dependencies generated from the articulation sets.

Theorem 4.12: For an acyclic hypergraph R, the articulation sets generate a cover for the multivalued dependencies generated by R.

Proof: According to Lemma 4.5, every separator of an acyclic hypergraph is contained in a hyperedge. Consider a separator S that is contained in hyperedge R_i, where the hyperedges are ordered by the maximal cardinality search. Clearly, $S \subsetneq R_i \cap R_j$ where $j = max(\beta(v) | v \in R_i$ and $\beta(v) < i)$, since otherwise S would not be able to separate R_i from the lower ordered hyperedges. Thus, S contains at least one articulation set. If S contain more vertices than the articulation set it contains these are superfluous. This means that the multivalued dependencies induced by S can be derived from the articulations sets contained in S by use of the union, augmentation, and complementation rule for multivalued dependencies. Thus, the articulation sets will produce a cover for the multivalued dependencies generated by the acyclic hypergraph. □

Note that the components produced by an articulation set S will correspond to the dependency basis of S. If we instead use a separator that is not minimal, we may not produce a cover for the set of multivalued dependencies.

Example 4.15: Let us say that the articulation set S produces components Y_1, Y_2, and Y_3. Let $A \in Y_1$, then SA will be a separator that produce the components $Y_1 - A$, Y_2, and Y_3. If $SA \rightarrow\rightarrow Y_1 - A | Y_2 | Y_3$ is added to the dependency set we see that $S \rightarrow\rightarrow Y_1 | Y_2 | Y_3$ can be implied from these multivalued dependencies. If, on the other hand, $S \rightarrow\rightarrow Y_1 | Y_2 | Y_3$ is added to the dependency set, then $SA \rightarrow\rightarrow Y_1 - \mathrm{A}$ can be implied from them. □

What remains is to find the right-hand sides of the multivalued dependencies. These right-hand sides are the connected components of H. Finding the connected components can be done by using depth-first search on the ordinary graph $G(H-S)$, where H is the hypergraph representation of the relation scheme and S is one of the articulation sets. The different components found will be the right-hand sides multidetermined by S. Unfortunately, the time complexity using the ordinary graph representation is quadratic in the size of the hypergraph it represents.

We will present a depth-first search restricted to hypergraphs that has complexity linear in the size of the hypergraph. This will give a faster way to find connected components than searching the ordinary graph. This search is a generalization of the maximal cardinality search for hypergraphs. The depth-first search for hypergraphs and the algorithm to find connected components for hypergraphs are described in Algorithm 4.1

Lemma 4.13: The connected components of a hypergraph can be found in time $O(n+p+\sum_{i=1}^{p} |R_i|)$.

Proof: The procedure search in Algorithm 4.1 is only called once for each hyperedge e in the hypergraph H. For each hyperedge e the list $V(e)$ is traversed once and similarly the list $E(v)$ is traversed once for each vertex $v \in V$. This means that step 2 in Algorithm 4.2 takes time $2 \cdot \sum_{i=1}^{p} |e_i|$, where p is the number of hyperedges in H. Step 1 of the algorithm takes time $O(n+p)$, where n is the number of vertices in H. This gives that Algorithm 4.1 has time complexity $O(n+p+\sum_{i=1}^{p} |R_i|)$, where relation R_i corresponds to the hyperedge e_i. □

The algorithm that takes an acyclic relational scheme and returns the set of induced multivalued dependencies is describe in Algorithm 4.3. The algorithm uses the maximal cardinality search to find the ordering of relations satisfying the running intersection property. While the search is done a list of the $p-1$ articulation sets are found. These articulation sets correspond to the right-hand sides of the multivalued dependencies that are needed to generate a cover for the multivalued dependencies induced by the relation scheme. After this step is completed the components found by removal of an articulation set from the scheme is found. The components correspond to the dependency basis or the left-hand sides of the articulation set or left-hand side of a multivalued dependency it corresponds to.

Theorem 4.14: Algorithm 4.2 finds a cover of multivalued dependencies generated by the hypergraph representing the acyclic relation scheme $R=(R_1,R_2,...,R_p)$ in $O(p \cdot (n+p+\sum_{i=1}^{p} |R_i|))$ time.

Proof: Theorem 4.10 states that an acyclic relation scheme has $p-1$ articulation sets that can be found in $O(n+\sum_{i=1}^{p} |R_i|)$ time. The connected components of a hypergraph are found in $O(n+p+\sum_{i=1}^{p} |R_i|)$ (see Theorem 4.13). Thus, it takes $O(p \cdot (n+p+\sum_{i=1}^{p} |R_i|))$ time to find the multivalued dependencies induced by the articulation sets. That these

Algorithm 4.1. Depth-First Search for Hypergraphs

Input: A hypergraph $H=(V,E)$ represented by the adjacency lists
$E(v)$ list of edges containing vertex v.
$V(e)$ list of vertices contained in hyperedge e.

Output: The connected components of the hypergraph $H=(V,E)$.

```
1. {Initialize the data structures}
    for all e∈E do
        unmark e;
    for all v∈V do
        unmark v;

2. {Find all components of hypergraph H}
    All_Components:=∅;
    Components:=∅;
    for all unmarked e∈E do
        search(e);
        All_Components:=All_Components∪{Components};
        Components:=∅;

3. {Return all the found components}
    return(All_Components)

procedure search(e)
    mark e;
    add e to Component;
    for every unmarked v∈V(e) do
        mark v;
        for every unmarked f∈E(v) do
            search(f);
```

Algorithm 4.2. Find Cover of Induced MVD's from Acyclic Scheme

Input: Acyclic hypergraph represented by $E(v)$, a list of edges containing vertex v, and $V(e)$, a list of vertices in hyperedge e.

Output: A cover for the mvd's induced by the acyclic scheme.

size(e) the number of numbered vertices in hyperedge *e*.
set(i) contains the edges with *i* numbered vertices.
$\beta(e)$ is the ordering assigned hyperedge *e*.
$\beta(v)$ is the lowest ordered hyperedge containing *v*.
$R(k)$ is the hyperedge numbered *k*.
$\gamma(e)=max(\beta(v) \mid v \in e$ and $\beta(v)<\beta(e))$.
j is the maximal number of numbered vertices in unnumbered hyperedge.
k is the number of the last selected hyperedge.
basis(S) is the dependency basis of *S*.

1. {Initialize the data structures}
    for $i \in [1,n-1]$ **do** $set(i):=\varnothing$;
    for all $e \in E$ **do** $size(e):=0$; $\gamma(e):=$ undefined; add *e* to $set(0)$;
    $j:=0$; $k:=0$;

2. {Find all the right-hand sides of the mvd's}
    while $j \geq 0$ **do**
        $k:=k+1$; $\beta(e):=$ k; $R(k):=e$; $size(e):=-1$;
        articulation _sets:=articulation_sets$\cup${numbered vertices of *e*};
        for all unnumbered $v \in V(e)$ **do**
            $\beta(v):=$k;
            for all $f \in E$ **and** $v \in E(f)$ **and** $size(f)$ **do**
                $\gamma(f):=$k; delete *f* from $set(size(f))$; $size(f):= size(f)+1$;
                if $size(f)<|f|$ **then** add *f* to $set(size(f))$; **else** $size(f):=-1$;
        $j:=j+1$;
        while $j \geq 0$ **and** $set(j)=\varnothing$ **do** $j:=j-1$;

3. {Find the left-hand sides belonging to each of the right-hand sides}
    for all $S \in$ *articulation _sets* **do**
        $basis(S):=$find_component$(H-S)$;

multivalued dependencies form a cover for all the multivalued dependencies induced from the scheme was proved in Theorem 4.12. □

According to Theorem 4.3, the multivalued dependencies generated will be equivalent to the join dependency $*(R_1,R_2,...,R_p)$. Thus, in addition to finding a cover for all the multivalued dependencies implied by the acyclic join dependency, we have also found a set of multivalued dependencies equivalent to the acyclic join dependency.

4.4. Summary

In this chapter we investigated the possibility of inducing dependencies that are satisfied among relations. This can be useful when a database designer wishes to restructure an existing relational database. The grouping of attributes into relations implicitly defines dependencies that can be useful in restructurting the database.

A relational database can be represented by a hypergraph. The multivalued dependencies induced by the datbase scheme are equiavlent to the separators in the hypergraph and the components they produce. For acyclic hypergraphs the minimal separators will be articulation sets. The problem of inducing dependencies from a relation scheme is also equivalent to finding multivalued dependencies induced by a join dependency. We presented an algorithm of $O(p \cdot (n+p+\sum_{i=1}^{p} |R_i|))$ time to find the multivalued dependencies equivalent to an acyclic join dependency $*(R_1,R_2,...,R_p)$, where p is the number of relations in the scheme and n is the number of attributes in the database.

CHAPTER 5

Query Design

The specification of relational database queries is usually considered a process totally separated from the logical design. Queries often impose implicit constraints to the natural grouping of attributes into a relation scheme, since the attributes in the query are related. Typically a designer may reject a design because the candidate scheme is not suited for the queries. Thus, the queries can be used to select a candidate design that supports the expected queries of the application well.

To make the query specification independent of a particular logical design the query is only specified by the attributes of the database. From this scheme independent query representation the exact query formulation for a particular candidate scheme must be found. The query is ambiguous if a candidate scheme has multiple query formulations for the query. An ambiguity can be a result of an erroneous query specification, a poor semantic specification, or simply an undesirable database scheme. Automatic query formulation can therefore be a useful tool for the design of queries as well as the logical design.

In Section 5.1, we will discuss the problem of finding a query formulation for a particular relation scheme, given a relation scheme independent query specification. We will also describe ambiguous queries and how such queries may be made unambiguous by modifying the logical design. In Section 5.2, we survey previous research in automatic query specification and query optimization. In Sections 5.3 and 5.4, we propose a new algorithm for finding all the join paths which can solve a query. In Section 5.5, we continue discussing the process of modifying the logical design to remove ambiguities. We conclude the chapter with a summary in Section 5.6.

5.1. Problem Definition

A goal of the logical design is to find a candidate scheme that is natural and simple to specify the queries on. Given a number of candidate designs the preferred design should suit the queries of the application. To do an appropriate selection we need to evaluate the designs with respect to the queries.

To ease the specification of queries a *universal relation interface* [Kort80] is used. In this interface, only the attributes in the query need to be specified and not the relations. Given the specification the design tool automatically finds the join paths of the query.

Definition 5.1: *Join Path of Query.* We say that a query has a join path if there is a path $R_i,...,R_k$ connecting all attributes in the query.

Determining the join paths of the query and displaying them to the database designer will indicate how good the design is for the application and how it compares to another design. We are interested in candidate schemes where the queries can be solved with few relations, implying few joins, and where few of the queries are *ambiguous*. Ambiguous is defined as:

Definition 5.2: *Ambiguous Query* [Ullm82]. We say that a query is ambiguous if for a specific candidate scheme $\mathbf{R}=\{R_1,R_2,...,R_p\}$ the query can be answered by the use of different set of relations of **R**.

Ambiguity is not the only problem, sometimes the design can not answer the query at all, since the attributes in the query are not related. In this case the query is solved by taking a cartesian product of some relations. Thus, it is also desirable that the relations used to form the query are connected.

To be able to solve a query specified using a universal relation interface we must find a set of relations $\{R_1,R_2,...,R_p\}$ from **R** that is needed to compute the query. Assume that we have the following candidates scheme for a banking application:

R_1=CUST_LIVES(Customer,Address)
R_2=CUST_ACC(Customer,Account)
R_3=CUST_LOAN(Customer,Loan)
R_4=ACC_BALANCE(Account,Balance)
R_5=LOAN_AMOUNT(Loan,Amount)
R_6=ACC_AT_BANK(Account,Bank)
R_7=LOAN_AT_BANK(Loan,Bank)

The design is illustrated by the diagram in Figure 5.1. Let us say that the following query is specified in a SQL like syntax:

select*(Bank)* **where** *Customer = 'Jones'*

There are two ways the query can be interpreted on the above banking database scheme and the designer should be made aware of this. The query could mean that the designer wants a list of all banks that customer Jones has an account in, or a list of all the banks that Jones has a loan in. There is also a possibility that the user wanted a list of all banks that Jones has an account or a loan in, i.e., the union of the previous solution. These possibilities can be indicated by showing the different relation sets that can produce an answer to the query. In this case:

(CUST_LOAN , LOAN_AT_BANK) and
(CUST_ACC , ACC_AT_BANK).

Using the query modification method suggested by Stonebraker [Ston75] the query

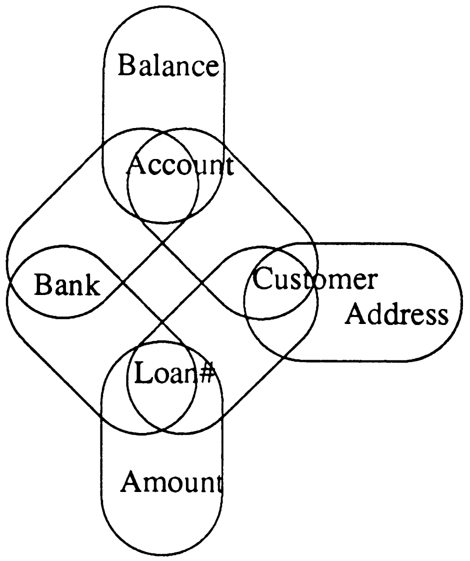


Figure 5.1. Banking Database Scheme

can be solved by one or both of the following SQL queries:
Query 1: **select** (t_2.*Bank*)
from CUST_LOAN t_1,
LOAN_AT_BANK t_2
where t_1.*loan* = t_2.*loan* **and** t_1.*Customer* = *'Jones'*
Query 2 : **select** (t_2.*Bank*)
from CUST_ACC t_1,
ACC_AT_BANK t_2
where t_1.*account* = t_2.*account* **and** t_1.*Customer* = *'Jones'*

The reason these two queries can produce different answers is because the join paths represent different semantic meanings and because the relations might contain *dangling tuples* [Ullm82] defined as:

Definition 5.3: *Dangling Tuple.* A dangling tuple is a tuple in a relation that does not match up with anything in a join with other relations.

If customer *Jones* only has a loan but does not have an account then the *Jones* tuple in CUST_LOAN is a dangling tuple since the tuple is lost when the relations is joined with another relation such as CUST_ACC. If CUST_ACC is the only relation containing *Jones* we see that the solution to Query 1 will be the empty set and to Query 2 the solution will be the bank where *Jones* has the loan. See Figure 5.2.

ACC_AT_BANK

Bank	Account
Chase	1
Chase	2

CUST_ACC

Account	Customer
1	Peters
2	Peters

LOAN_AT_BANK

Bank	Loan#
First	3
First	4

CUST_LOAN

Loan#	Customer
3	Jones
4	Peters

Figure 5.2. Instance of Banking Database

Note that if we falsely defined the query over the cyclic relation set (CUST_ACC, ACC_AT_BANK, LOAN_AT_BANK, CUST_LOAN) believing that we would solve the union of the two queries, we see that the answer would be the empty set rather than the bank where *Jones* has a loan. Cycles in the join path will always exclude dangling tuples from the answer, while acyclic join paths will include them.

Some researchers avoid the problem of dangling tuples by asserting the *universal relation assumption* [Ullm82]. The universal relation assumption states that at all times the database relations are the projections of a single universal relation. We find that this is an unrealistic assumption since one of the main purposes of normalization is to allow dangling tuples without having update and insertion anomalies. Note that the universal relation assumption simplifies the problem of finding the query join paths since all paths will produce the same answer.

In addition to avoiding cycles one must also avoid traversing join paths that are lossy. It is known that a join of two relations that cannot be joined losslessly will produce tuples that originally did not exist in the database. Formally, a lossless decomposition is defined as:

Definition 5.4: *Lossless Join Decomposition* [Ullm82]. If **R** is a relation scheme decomposed into $R_1, R_2, ..., R_p$, and D is a s set of dependencies, we say the decomposition is a lossless join decomposition if for every relation instance **r** for **R** satisfying dependencies D:

$$\mathbf{r}=\pi_{R_1}(\mathbf{r})*\pi_{R_2}(\mathbf{r})*\cdots*\pi_{R_p}(\mathbf{r})$$

that is, **r** is the join of its projections onto the R_i's.

Given dependencies, the two relations R_1 and R_2 will have a lossless join when $R_1 \cap R_2 \rightarrow R_1 \mid R_2$ or $R_1 \cap R_2 \rightarrow\rightarrow R_1 - R_2 \mid R_2 - R_1$ holds.

Example 5.1: Let both customer *Jones* and *Peters* have a regular bank account and loan account at *Chase*. Instances of the relations ACC_AT_BANK, LOAN_AT_BANK, and CUST_LOAN are given in Figure 5.3. If these relations are joined to find an answer to **select** (Bank) **where** Customer=*Jones* one will discover that *Jones* suddenly has two bank accounts rather than one. □

We see that to answer a query properly one can not simply gather relations that span the attributes in the query. A join path of the query must be lossless in order to produce a correct answer. Also, the underlying relational database onto which the queries are mapped should always have a lossless join. This can always be accomplished since given a set of attributes for the universal relation and a set of associated functional dependencies, one can decompose into a relation scheme such that the universal relation is a lossless join of all the decomposed relations [Fagi82].

We have seen that cyclic relation schemes can make the queries ambiguous. Acyclic queries, on the other hand, have a unique solution to any query [Maie83a]. Some researchers believe that most applications can be expressed with acyclic relation schemes and that cyclicity is undesirable [Lien82, Scio82]. There are several reasons that cycles occur in schemes. One is that an attribute is used to represent two or more different properties of the application. Another reason, is that the logical dependencies are incomplete or wrong, such that they produce undesirable candidate schemes. Two ways of making cyclic schemes acyclic have been suggested:

1. Introducing new attributes [Scio82,Fagi82] or
2. Introducing new relations [Chase81].

Applying the approach to introduce new attributes to the banking database would be to either replace the attribute Customer by the attributes Loan_Customer and Account_Customer or split attribute Bank into Loan_Bank and Account_Bank (Figure 5.4.a). In both cases, the resulting design will have a unique join path for the formerly ambiguous query.

ACC_AT_BANK

Bank	Account
Chase	1
Chase	2

LOAN_AT_BANK

Bank	Loan#
Chase	3
Chase	4

CUST_LOAN

Loan#	Customer
3	Jones
4	Peters

Figure 5.3. Instance of Banking Database

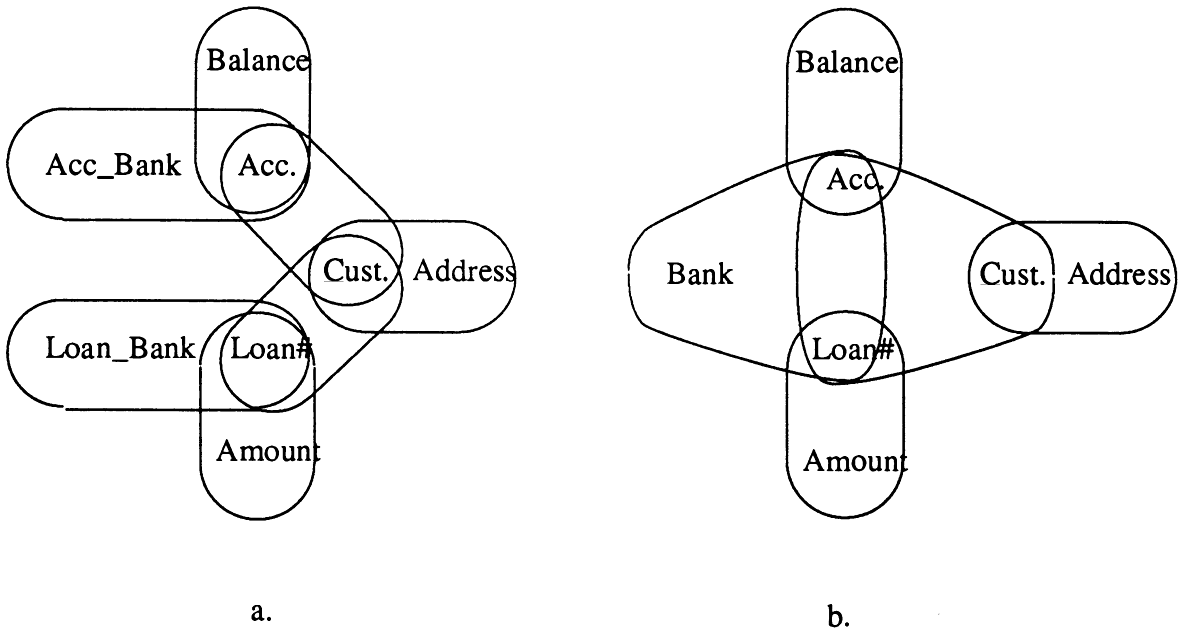


Figure 5.4. Scheme Modifications

The above modification is not desirable if the designer actually wanted to list banks that a person has a loan or an account in. In this case, suggesting new relations that will make the candidate relation scheme acyclic will be more appropriate. Applied to this example, the cycle of four relations can be substituted by one of the two following sets of relations: {(Bank Loan Customer) (Bank Account Customer)} or {(Bank Account Loan) (Cust Account Loan)}. Chosing the second set will result in the design showed in Figure 5.4 b.

This type of user interface introduces two problems that will be studied in further detail, namely:

1. Automatic join path finding.
2. Automatic presentation of attributes and/or relations that can make the database model provide a unique answer to the query.

We have seen that to find a join path for a query we must assure that no lossy paths are traversed, that there is no cycle in the path, and that there are no unnecessary relations.

5.2. Previous Research

It is well known that the join operation is the most expensive operation in query processing and that arbitrary queries ranging over multiple relations can have many unnecessary joins. Elimination of unnecessary join operations has been studied in context with query optimization, query execution in universal relation database systems, and relational database systems with views. Several methods to find the best join path or minimize the number of joins for a query have been proposed. In this section, we survey the algorithms that find join paths, based on (a) the functional dependencies (Section 5.2.1), (b) the query tableau (Section 5.2.2), and (c) the concept of maximal object (Section 5.2.3).

5.2.1. SPJ Access Paths

Carlson and Kaplan [Carl76] investigated the possibility of finding access paths for relation independent select, project, join queries (SPJ-query) on the form:

select $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$
where A_1 *op* c_1 **and/or ... and/or** A_j op c_j

The functional dependencies in addition to the actual relations in the database are used to determine an access path for the query. A path $P=\{p_1,p_2,\ldots,p_n\}$ with p_j on the form $R_j.A_k$, is an access path if one of the following conditions holds for p_j for $j=1,\ldots,n-1$:

1. $p_j \rightarrow p_{j+1}$
2. $p_j \leftarrow p_{j+1}$
3. there exist two relations R_1 and R_2 such that $p_j=R_1.A$ and $p_{j+1}=R_2.A$ and $\pi_A(r_1) \cap \pi_A(r_2) \neq \emptyset$ where r_1 and r_2 are relation instances of R_1 and R_2.

The last condition states that the two relations must contain values that can joined.

In the case where the where-clause contains a boolean expression containing a conjunction or disjunction of simple predicates the query is decomposed into several simple queries whose access path P is a simple path and not a tree. This means that the target attributes must be contained in one relation. The final answer of the query is found by taking unions and intersections of the simple query answers, depending on the conjunctions and disjunctions in the where-clause. Carlson and Kaplan do not give an account of the complexity of the method. But the method is clearly dependent on the size of the relation scheme and its functional dependencies as well as the size of the database.

5.2.2. The Tableau Method

The *tableau method* by Aho, Sagiv, and Ullman [Aho79c] optimizes SPJ-query by reducing the number of joins in the expression. The tableau is a two dimensional array. The first row of the tableau lists the attributes in the database. The next row, the summary, contains symbols "a" in the positions corresponding to the attributes in the target list. The other positions are blank. Each relation is assigned a row in the tableau, with symbol "b" placed in the positions corresponding to the attributes in the relation. If the query specifies a specific constant for an attribute the b is substituted with that constant, and if the relation contains an attribute of the target list the b is relaced by an a. If the query contains some conditions relating the variables in the tableau these are added as *constraints* to the tableau. Note that instead of placing a constant in the tableau a constraint such as b_i=$constant_k$ can be added to the tableau.

Example 5.2: The tableau of the query **select**(*Bank*) **where** *Customer='Jones'* over the relations in the banking database is given in Table 5.1. □

Table 5.1. Tableau Representation

Relation	Bal.	Acc.	Cust.	Addr.	Ln#	Amt.	Bnk
Summary							a1
R_1			Jones	b1			
R_2		b2	Jones				
R_3			Jones		b3		
R_4	b4	b2					
R_5					b3	b5	
R_6		b2					a1
R_7					b3		a1

A tableau with n relation rows indicates that $(n-1)$ join operations are needed in the query expression. Therefore, to minimize the number of joins in the query expression the number of relation rows in the tableau should be reduced. This is done by finding a tableau T_2 which is equivalent but smaller than the original tableau T_1. The following theorems and definition are given by Ullman [Ullm82].

Theorem 5.1: Tableau T_1 is *contained* in tableau T_2 if and only if there is a symbol mapping h from the symbols of T_2 to those of T_1 such that:

1. h applied to the summary of T_2 is the summary of T_1,
2. h applied to each row of T_2 (h maps any constants to themselves) is a row of T_1 corresponding to the same relation, and
3. each constraint in T_2 is implied by the hypothesis that the constraints of T_1 hold for the symbols to which h maps the symbols of T_2.

Definition 5.5: *Equivalence of Tableaus.* Two tableaus T_1 and T_2 are equivalent if T_1 is contained in T_2 and T_2 is contained in T_1.

Theorem 5.2: If T_1 is any tableau, there is a minimal equivalent formed tableau T_2 found by deleting zero or more rows of T_1.

Example 5.3: Consider the tableau of Example 5.3. The tableau given in Table 5.2 is its minimal equivalent tableau, where the mapping functions between the tableaus are:

Mapping for T1 to t2	Mapping from T2 to T1
h(a1)=a1	g(a1)=a1
h(b1)=""	g(b2)=b2
h(b2)=b2	g(b3)=b3
h(b3)=b3	g("Jones")="Jones"
h(b4)=""	
h(b5)=""	
h("Jones")="Jones"	

Table 5.2. Minimized Tableau Representation

Relation	Bal.	Acc.	Cust.	Addr.	Ln#	Amt.	Bnk
Summary							a1
R_2		b2	Jones				
R_3			Jones		b3		
R_6		b2					a1
R_7					b3		a1

The minimization of a tableau is NP-complete in the general case [Ullm82], but the method has been shown to be polynomial for some subclass of SPJ-queries. The main argument for using the tableau method is that in general the number of relations involved in the query (or the number of joins) will be so small that using this optimization method is practical.

5.2.3. Maximal Objects

The tableau method works well as long as the database scheme is acyclic. When the scheme is cyclic the minimized tableau can contain several solution paths, as we see from the above example. Thus, for cyclic schemes the tableau method will not be able to distinguish the different solution paths and dangling tuples may be excluded from the answer of the query. If lossy paths exist the tableau method will include these lossy paths which is not desirable. Researchers have proposed ways of finding a subset of a relation scheme that contains no lossy paths. These sets are given to the tableau method for further optimization and each of the sets produce a possible partial answer to the query.

Maier and Ullman introduced the concept of *maximal objects* [Maie83b] to compute the correct answers to queries to universal relation databases. The concept is defined as:

Definition 5.6: *Maximal Object.* A maximal object is a subset of the relation scheme that can be joined losslessly.

The goal of the maximal object is not to break the cycles in the relation scheme but to provide the tableau with relations containing no lossy paths. The algorithm for calculating a maximal object is as follows: Accumulate relations one at a time such that the join between the accumulated relations and the new relation is lossless. By starting this computation from all relations all the maximal objects will be found. A maximal object that is a subset of other maximal object is deleted. The solution of a query is the union of all the answers achieved from the maximal objects containing the all attributes relevant to the query. To determine whether a join is lossless or not the functional dependencies and multivalued dependencies in the database are used. The time complexity to find all maximal objects is polynomial in the size of the relation scheme.

Example 5.4: Consider the relation scheme of Example 5.1. Assuming that all the joins are lossless there will be only one maximal object consisting of the whole relation scheme. □

The above example shows us that the notion of maximal object does not solve the problem of multiple join paths. The maximal object does not even completely exclude the possibility that some of these paths are lossy. Say that relations LOAN_AT_BANK and ACC_AT_BANK from Example 5.1 have a lossy join, the maximal object will still be the whole relation scheme since they both can be joined losslessly to the other relations. We see that the cycle must be broken otherwise lossy joins can be involved in the query execution or dangling tuples will be excluded from the answer.

Zhang and Mendelzon [Zhan83] defined the *acyclic maximal object* where the maximal objects were required to contain no cycles. They define acyclic maximal objects from entity-relationship diagrams. It is assumed that each lossless paths can produce a different answer to the query and the complete answer to the query is found by taking the union of all the answers found over the acyclic maximal objects that contain the attributes of the query. The following example will show that this approach can result in unnecessary computation.

Example 5.5: The relation scheme of Example 5.1 has four acyclic maximal objects. Each acyclic maximal object has one relation from the cycle removed. To find the solution of the query:

> **select***(Bank)*
> **where** *Customer* = *'Jones'*

all the four acyclic maximal objects must be used. Clearly this is not a very satisfying solution since two of them would be sufficient to compute the answer. □

Pawha and Arora [Pawh85] base their method on Maier and Ullman's maximal object but handle cyclic maximal objects separately. To avoid the problems of lossy paths and dangling tuples of cyclic maximal objects they compute the *cyclic subset maximal objects* (CSMO) that contain each of the cycles in the maximal object. For each of these cyclic subset maximal objects, maximal *individual paths* are computed. Individual paths are defined as:

Definition 5.7: *Individual Paths.* An individual path is a path in a CSMO that is acyclic and lossless.

Pawha and Arora assume that if the individual paths have a solution, then these solutions are identical. Thus an individual path has a solution or no solution at all. This simplifies the rather complex task of computing the answer to the query from the cyclic subset maximal objects and the individual paths using the tableau method repeatedly. The order of Pawha and Arora's method depends on the number of cycles in the hypergraph representing the relation scheme. A complexity analysis of the method was not given and a query as a path in this graph.

5.2.4. Steiner-Trees

Under the context of mapping a query specified in a natural language to a query specified on an ER-model, Motro and also Lin make the assumption that the minimal subtree or Steiner-tree is sufficient to answer the query [Motr86, Lin89]. A ER-model is represented as an ordinary graph.

Definition 5.8: *Steiner-Tree.* Let $G=(V,E)$ be a connected graph where V is the set of vertices and E the set of edges. Given a set of vertices $V^o \subseteq V$, the Steiner-tree is a subtree T of G such that:

1. the vertices V^o are interconnected by edges in T.
2. the total length of the edges in T is minimal.

The Steiner-trees will represent the optimized query which is assumed to be the query the database designer wanted to design. The Steiner-tree problem is known to be NP-complete [Gare79].

5.3. Query Disambiguation

Finding join paths for a query can be useful to do during the logical design. In this context we are more interested in the semantics and structure of the database scheme than in the actual execution of the query. As described in Section 5.1, showing the different join paths of an ambiguous query can help the designer to evaluate the query as well as the candidate scheme. The methods using maximal objects and individual paths described in the previous sections, are usually computed once and then stored and used by the database system. For a design tool this will not be the appropriate approach since the design varies frequently and it would also be better to use a method that does not use the tableau method at any stage.

From the previous sections, we see that to find the join path producing a correct answer to the specified query we must make sure that the paths are lossless, that no cycles occur, and that unnecessary relations are removed. More formally the problem can be stated as:

> For the given set of attributes V^o involved in the query specification, find all connected subhypergraphs of **R**= $\{R_1, R_2, \ldots, R_p\}$ such that for each subhypergraph **R'** contains V^o and there is no lossless paths and that **R'** is acyclic. **R'** should be minimized such that for any $R_i \in$ **R'**, **R'**$-R_i$ is disconnected or $V^o \not\subset$ **R'**$-R_i$.

This problem definition states that we are searching for relation sets whose hypergraph representations have properties similar to trees. If we only had binary relations, relations containing only two attributes, the hypergraph would be a ordinary graph and our problem would be reduced to finding a subtree of a spanning tree containing the attributes V_o.

Restricting the join path to be a the Steiner-tree as Motro and Lin [Motr86,Lin89] do, is not useful for the purpose of finding anomalies. The minimal join path will often be the desired answer in a well designed database, but during the design phase the design can have some undesirable properties such that this is not the case. Consider again the banking database now modified to also contain the social security number of a customer as prepresented in Figure 5.3. If

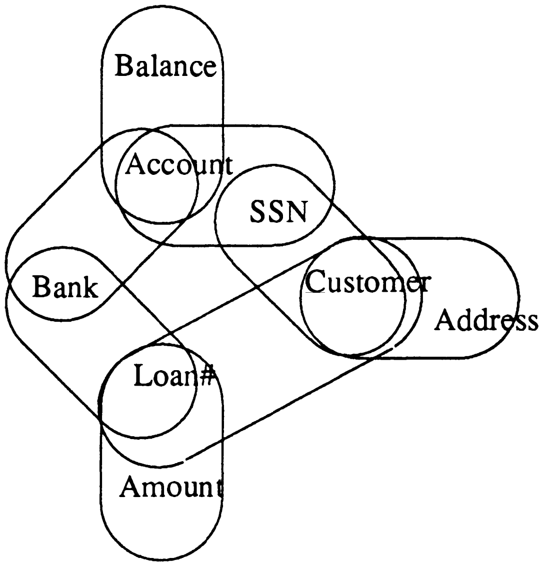


Figure 5.5. Modified Banking Database Scheme

we restricted the possibility of solutions of the query from Example 5.4 to be the minimal join paths, there is only one solution but the solution no longer includes the bank accounts of *Jones*. By displaying the different paths we see that the designer can be made aware of another possible solution to the query. As long as the paths are lossless the answer is considered correct, even if it is not the shortest possible solution. Based on these solutions, the designer can make a decision whether the query is correct or whether the candidate scheme should be modified. In our above example the designer is likely to change the design such that the Social Security Number (SSN), Customer, and Address form one relation.

In the general case, a relational scheme does not only contain binary relations and we must therefore generalize the concept of spanning trees to hypergraphs.

Definition 5.9: *Hyper Spanning Tree.* A hyper spanning tree $S(V',E')$ of a hypergraph $H(V,E)$ is an acyclic hypergraph such that for every $e \in E-E'$ added to S will make the hypergraph cyclic or disconnected.

Note that adding the constraint that the paths are lossless, the above definition would be an acyclic maximal object.

Unfortunately, there are some cyclic hypergraphs whose hyper spanning trees do not contain all the vertices that are essential for building paths (i.e., the vertices excluding the knobs). This means that some vertices can not be reached unless there is cycle in the join path. This is again a anomaly with the design, since we know that cycles will exclude dangling tuples from the answer of the query.

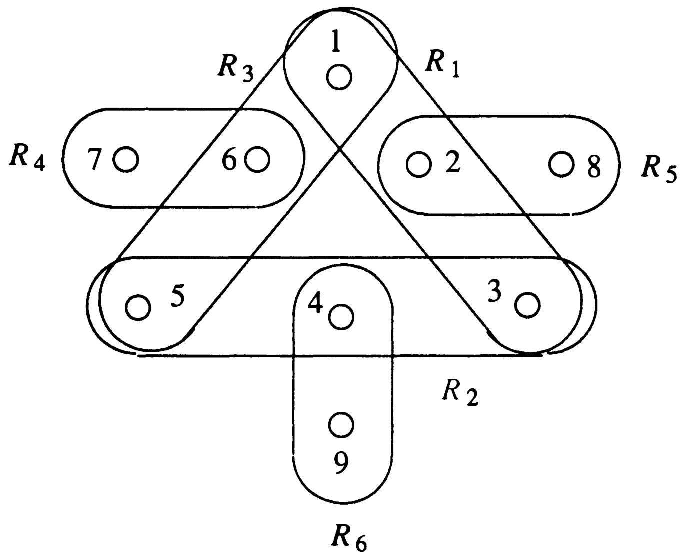


Figure 5.6. Cyclic Hypergraph

Example 5.6: Consider the hypergraph H=({1 2 3 4 5 6 7 8 9},{{1 2 3},{3 4 5}, {5 6 1},{6 7},{2 8},{4 9}})=(R,{R_1,R_2,R_3,R_4,R_5,R_6}) given in Figure 5.6. There is no hyper spanning tree that can span all the vertices. The hypergraph has the three following hyper spanning trees. S_1=({1 2 3 5 6 7 8},{R_1,R_3,R_4,R_5}) S_2=({1 2 3 4 5 8 9},{R_1,R_2,R_5,R_6}) S_3=({1 3 4 5 6 7 9},{R_2,R_3,R_4,R_6}). □

To solve our problem of finding join paths for queries we need not find whole hyper spanning trees, but only parts of the hyper spanning trees that span the vertices V^o that occur in the query. We are thus interested in a sub hyper spanning tree defined as follows:

Definition 5.10: *Sub hyper spanning tree*. Given a hypergraph $H(V,E)$ and $V^o \subseteq V$. A sub hyper spanning tree S_{V^o}(V'',T) is an acyclic sub hypergraph of a hyper spanning tree S of H that includes all vertices of $V^o \subseteq V'' \subseteq V$ and $T \subseteq E$. S_{V^o} is minimal such that removal of any articulation set will disconnect some vertices of V^o.

We have the following lemma for sub hyper spanning trees.

Lemma 5.3: Let S_{V^o}(V'',T) be a sub hyper spanning tree of a hypergraph $H(V,E)$, where $V^o \subseteq V'' \subseteq V$ and $T \subseteq E$. Then if an edge $e \in E-T$ is added to S_{V^o} the resulting hypergraph will either be:

(a) cyclic,
(b) disconnected, or
(c) contain an articulation set that does not disconnect the vertices in V^o.

Proof: Clearly, if an edge e not in S is added to S_{V^o} the resulting hypergraph can either be cyclic or disconnected. That $S_{V^o} \cup e$ is not cyclic or nor disconnected contradicts that S was a hyper spanning tree. Now, if we add a hyperedge e from S not in S_{V^o}, either $S_{V^o} \cup e$ is cyclic since a sub hypergraph of an acyclic hypergraph can be cyclic or $S_{V^o} \cup e$ is disconnected if $S_{V^o} \cup e = \varnothing$ or $S_{V^o} \cap e$ has the articulation set $S_{V^o} \cap e$ that does not disconnect any vertices of V^o. That $S_{V^o} \cap e$ disconnects some vertices of V^o contradicts S_{V^o} spanned all the vertices of V^o. □

This lemma gives a constructive way of eliminating paths in the initial hypergraph that will not be used to span the vertices V^o. Components, found by removing an articulation set, that contain no vertices of V^o can be removed. No path between any two vertices of V^o is needed1 to traverse any of these components.

The algorithm to find articulation set of acyclic hypergraphs, presented in Chapter 4, can not be used since it is restricted to work on acyclic hypergraphs. For cyclic hypergraphs we must use a more general approach which will be to check whether every intersection of two hyperedges is an articulation set. For every articulation set find the components its removal produces and remove the components that do not contain any of the vertices in V^o. Testing whether a hypergraph is connected or disconnected can be done by using Algorithm 4.2 as described in Chapter 4. The time complexity of finding the articulations sets and removing the components not containing any vertices from V^o is $O(p^2 \cdot \sum_{i=1}^{p} |R_i|)$ time in the worst case.

After finding the sub hypergraph with unnecessary components removed, we need to find the sub hyper spanning trees that will correspond to the possible join paths for the query. We do this by finding the hyper spanning trees of the hypergraph and then minimizing these by using an algorithm due to Tarjan and Yannakakis [Tarj84] that will be discussed later in the section.

The algorithm for finding all hyper spanning trees, presented in Algorithm 5.1, is based on an idea from McIlroy's algorithm for generating all spanning trees for ordinary graphs [McIl66]. Our algorithm generalizes this algorithm for hypergraphs. Finding all spanning trees of a ordinary graph is of exponential time in the number of vertices of the graph, and this is also the case for hypergraphs. The high complexity is a result of many solutions due to the cycles in the graph. We believe that relation schemes for real world applications are not inherently cyclic, but almost acyclic. Thus, in practice there will be a limited number of solutions.

In Algorithm 5.1, the hyper graph is represented by the sets $J[i]$ where i is a hyperedge and $J[i]$ is the set of hyperedges that i intersect. The set *Tree* contains all the hyperedges in the growing hyper spanning tree. The set *Tree-vertices* contain all the vertices in the *Tree*. The boundary set *Border* contain hyperedges adjacent to *Tree* and are hyperedges that can be selected next. When *Border* is empty a hyper spanning tree has been found. The correctness of the algorithm is based on the following theorem.

Theorem 5.4: Let J, *Tree*, *Tree_vertices*, and *Border* be as defined above and R_i the vertices of hyperedge i. Hyperedge $i \in Border$ can be added to *Tree* without making it acyclic if and only if at least one of the following three conditions hold:

1. $|Tree \cap J[i]| = 1$
2. $Tree\_vertices \cap R_i \subseteq R_j$ *where* $j \in Tree$
3. $R_k \cap R_j \subseteq R_i \; \forall k,j \in Tree \cap J[i]$ where $k \neq j$.

Proof: A hypergraph is acyclic if it contains no cycles or if every cycle of size three or more contains a chord (Definition 4.12). Clearly, if condition 1 holds the hyperedge i is connected to only one hyperedge of *Tree* and no cycle will be introduced by adding i to *Tree*. If hyperedge i intersects a number of hyperedges we introduce a cycle if hyperedge i connects two hyperedges that already are connected. If condition 2 holds we see that the intersection of hyperedge i and *Tree* is contained in one hyperedge, thus the hypergraph remains acyclic after adding i. If the intersection is not contained within an hyperedge we must make sure that hyperedge i is a chord. By Definition 4.12 i is a chord if two intersecting hyperedges intersecting i is contained in i, giving condition 3. □

When a hyperedge is found to introduce a cycle it can not simply be removed from the hypergraph representation. A hyperedge that at some point was rejected because it introduced a cycle can be added at a lower recursion if a hyperedge that is

Algorithm 5.1. Algorithm for Finding Hyper Spanning Trees.

Input: A hypergraph $H(V,E)$ represented by J and R and the variable *Border* set to be hyperedges containing some vertex which is not a knob. Variables *Tree* and *Tree_vertices* are initially empty.

Output: The hyper spanning trees of $H(V,E)$.

```
procedure span(Tree,Tree_vertices,Border,J)
    if (|Border|≥1) then
        for all edge i ∈ Border do
            <remove i from Border>
            inter:=J[i] ∩ Tree;
            inter_vertices:=R_i∩Tree_vertices;
            if (|inter|>1) then
                cycle:=true;
                for j∈ inter do
                    if (inter_vertices⊆R_j) then cycle:=false;
                if (cycle) then
                    cycle:=false;
                    for j∈ inter do
                        for k∈ inter and j≠k do
                            if R_j∩R_k⊈R_i then cycle:=true;
                if (not cycle) then
                    J[i]:=J[i] & ¬ Tree;
                    span(union(Tree,i),Tree_verticesR∪R_i,
                        new_border(Border,i,J),J);
            end if
        end for all
    else < print hyper spanning tree found >
```

the cycles chord has been added to *Tree.*

The *Border* should initially be a set of hyperedges containing one specific vertex of V^o that is not a knob. If all vertices of V^o are knobs one should select a vertex that represents an attribute that functionally determines some of the attributes occurring in the query. The procedure new_border adds to *Border* the hyperedges intersecting i not already in *Tree*. The procedure adds these hyperedges such that a depth first search traversal of the hypergraph is assured.

The algorithm as described has not taken lossless joins in consideration. This can be incorporated to the new_border procedure so that *Border* only contains hyperedges that can be joined losslessly with the hyperedges in *Tree* or done while testing whether the addition of an edge i will leave the hypergraph acyclic. The testing for losslessness is equivalent to testing whether *inter-vertices* functionally or multi determines *Tree_vertices–i* or *i-Tree_vertices.*

Query optimization has been found to be simple in the case where the database scheme is acyclic. Tarjan and Yannakakis [Tarj84] solves the join path finding problem for acyclic schemes by a *selective reduction* algorithm. This algorithm can be used to find the unique sub hyper spanning tree for the query given the hyper spanning tree. Their algorithm can be described as follows:

1. First mark the attributes X that are involved in the query.
2. (i) delete unmarked attributes that occur in only one relation.
   (ii) delete a relation that is contained in another relation.

After this reduction is completed we are left with a collection of attribute sets $Y_1,...,Y_k$ where each Y_i is a subset of a relation. For each Y_i we find all the relations that contain Y_i, project these relations onto Y_i and take the union of these projections, producing a single relation for Y_i. Then we take the join of the relations obtained for $Y_1,...,Y_k$. This process assures that the dangling tuples are included in the answer. Tarjan and Yannakakis give a linear time algorithm to do the selective reduction.

The method for finding the join paths of an acyclic relation scheme can be summarized as:

1. Reduce the cyclic hypergraph representing the database scheme by removing unnecessary paths.
2. Find the acyclic subhypergraphs or hyper spanning trees that contain the vertices V^o corresponding to the attributes in the query.
3. Do a selective reduction on each of the acyclic subhypergraphs found.

For some cases this method will not be able to find an answer. This will be considered as an anomaly either with the query itself or the candidate design. One case is that the query might not have a join path between attributes in the query, meaning that the query must be solved by a cartesian product, the join path is not lossless, or the join path contains a cycle. These three cases are considered undesirable and in such cases the tool should notify that there is a problem with the design and the query.

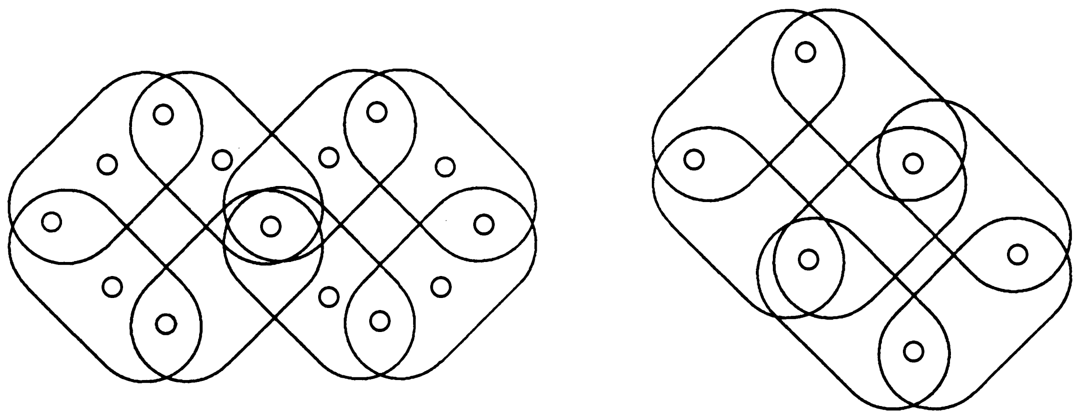
5.4. Correcting Ambiguous Designs

In this section, we will discuss the problem of correcting schemes from being cyclic to becoming acyclic a little further. We have seen that queries over cyclic schemes can be ambiguous. To make a query unique the cycles must be modified. It is not necessary to make the whole scheme acyclic: only the part of the scheme involved in the query.

The problem of correcting designs to make the queries unambiguous has not been studied much. Chase has studied the approach of adding or substituting relations in the scheme [Chase81]. Her method will, generally speaking, substitute a cycle with one relation containing all the attributes in the cycle. This is clearly a very crude approach and we will analyse the problem further.

An acyclic hypergraph can have cycles that are independent as shown in Figure 5.4 (a) or cycles that intersect as shown in in Figure 5.4. (b). The independent cycles can be modified without affecting the other cycles which is not always the case for intersecting cycles.

Making the cycles of the hypergraph acyclic can be done in a number of ways as we indicated in the beginning of this chapter. The larger the cycle, the more possibilities exist. But in essence one must either add hyperedges that are chords or merge several relations into one relation. With large cycles the number of combinations can make them impractical to display to the database designer. One way would be to display the simplest and crudest solution, that is replacing each cycle with one relation. Chase was not able to handle the case of intersecting cycles properly since the method would merge the two cycles into one relation. After displaying the possibilities, the designer should then decide whether other solutions should be displayed. Unless the cycles are small this can tend to be a time consuming task and is probably



(a) Independent Cycles (b) Intersecting Cycles

Figure 5.7. Cycles in Hypergraphs

not very helpful. The best solution would be to suggest to join the relations in a cycle that contain attributes of a query, but this might not be the desired solution. The designer could suggest an improvement of the design on the basis of the query and its multiple join paths

5.5. Summary

In this chapter, we presented a method to automatically find the exact query formulation of a query specified in the universal relation interface. Queries specified over a cyclic relation scheme can often be ambiguous. In this case, our algorithm will find the possible formulations for an ambiguous query. The algorithm is exponential in the number of attributes of the relation scheme, but it is linear in the number of hyper spanning trees of the hypergraph corresponding to the relation scheme.

An ambiguity in the query or an undesirable join path can be a result of an erroneous query specification, a poor semantic specification for the database application, or simply an indication that the current database design is not natural. We briefly discuss ways in which the relation scheme can be modified in order to produce a better scheme. Unfortunately, because there may be many alternative schemes, this improvement process cannot be fully automated. Guidelines should be provided by the database designer in an interactive design dialogue.

CHAPTER 6

Implementation and Performance Issues

In this chapter, we address specific issues related to feasibility, ease of use, and performance of automatic database design tools. We describe our experience with implementing a prototype design tool, Design By Example (*DBE*), and present a detailed performance study of the dependency inference algorithms that are used in the tool.

This chapter is organized as follows. In Section 6.1, we describe the design goals of *DBE*. In this overview, we emphasize the characteristics of *DBE* that differ from other design tools; the use of example tables and queries for representing logical dependencies and the presentation of alternative database schemes that the designer can select. In Section 6.2, we describe the *DBE* design screens used in different phases of the logical design. In sections 6.3 and 6.4 the internal structures and memory management of *DBE* is described. In Section 6.5, we give a short description of the two inference algorithms proposed by Räihä and Mannila [Mann85, Mann87]. We present a number of optimizations and heuristics, that improved their performance on an extensive set of examples. In Section 6.6, we perform experiments on the different inference algorithms on a number of different relations to investigate the feasibility of using example tables to specify functional dependencies. We conclude the chapter with a summary in Section 6.7.

6.1. Overview of the DBE Interface

Design By Example (*DBE*)is an expert tool for specifying and designing a relational database and its associated queries. The *DBE* prototype makes the knowledge in relational design theory automatically and transparently available to database designers. The primary goal in the development of *DBE* was to provide a system that can tolerate, elucidate, and correct poorly formed relational database specifications by using example tables and example queries. The system supports an interactive design process, which takes the database designer through a structured sequence of *Design Screens*. The screens visualize design information and provide menu selections to invoke specialized editors and design algorithms. An underlying storage subsystem provides access to a design catalog containing the attributes, dependencies, relations, tables, and queries participating in a design session.

The design expertise embedded in *DBE* addresses two important problems that current database design tools often overlook. The first problem is that the database designer is usually not able to specify the semantics of the database or the queries correctly and completely in one step. Thus, a good design tool should support an iterative design process and provide a way to verify and correct an initial specification. For that purpose, *DBE* represents design information with examples that the designer can examine and modify. The second problem is that alternative database schemes often exist of which the designer is not aware. During a conventional design process, one out of possibly many candidate schemes is often selected, without weighing whether it is natural for representing data or for formulating queries. In contrast, *DBE* generates multiple designs for the same database and lets the designer examine and evaluate them in conjunction with queries.

Anomalies resulting from an incorrect specification of the logical dependencies often appear in a relational scheme that is produced by a conventional design tool or chosen by the designer in an ad hoc manner. Even for the best understood dependencies, the functional dependencies, omissions or errors will often occur when the database designer is asked to specify them all at once. To cope with this problem, *DBE* can automatically infer functional dependencies from an example table containing sample data. This inference process can be very useful when a sample of real data is available for the database application being designed. In this situation, instead of requiring the database designer to specify the functional dependencies, these could be automatically inferred from the sample data. Alternatively, if the designer prefers to specify the dependencies directly, but needs to verify this specification, *DBE* will generate an example table that satisfies exactly these dependencies. The data values used in generating the example table are selected from domains defined by the database designer. Thus, by examining the familiar data in the table, the designer may uncover inconsistencies. In particular, for any dependency that was omitted in the initial specification, the example table will contain a pair of "real-like" tuples violating this dependency.

Other database anomalies may result when the designer chooses a scheme that is not sufficiently normalized or that does not have other desirable properties such as lossless joins or preservation of dependencies [Ullm82]. To avoid these anomalies, *DBE* generates database schemes satisfying these properties, given logical dependencies, or checks whether these properties hold for a scheme that was manually constructed by the designer.

Ambiguities in a relational query may result from an unnatural database scheme, from an unfriendly query language, or simply from a user error. Often, these ambiguities could have been avoided by making the database designer aware of an alternative scheme for the database and by providing a tool for validating the formulation of queries. *DBE* provides the database designer with the option to reject or modify a candidate database scheme if it is difficult to formulate a desirable set of queries against that scheme. Thus, the design and acceptance of a database scheme is fully integrated with the design of the queries. Furthermore, the design tool can be used on an operational database to update and modify the database scheme if new data and new applications make the existing scheme incomplete or inefficient.

Inferring functional dependencies from example tables are one of the most central and complex operations in the design tool *DBE*. Unfortunately, at first sight, algorithms for dependency inference look discouragingly slow. The problem is inherently exponential in the number of attributes, and, for each candidate dependency, requires comparing every pair of tuples in the relation. Two algorithms were recently proposed [Mann85, Mann87] that attempt to improve the naive approach to dependency inference by structuring and utilizing global information in the relation. The availability of a prototype for *DBE*, where we have paid careful consideration to efficient data structures and memory management, has enabled us to systematically experiment with automatic dependency inference. We implemented the above algorithms and performed an extensive performance study, which demonstrates the feasibility of inferring functional dependencies from an example relation (see Section 6.6 below).

6.2. The DBE Design Screens

The design session with Design By Example takes the database designer through a sequence of design screens, where the current design information is displayed and menu selections are provided. A transparent history mechanism keeps track of the work accomplished during the session. Certain screens are *Viewers* that display either design information created automatically or input by designer. Other screens are *Editors* that are used to input or update the design information.

The initial screen invoked by *DBE* is the *Catalog Viewer*. The Catalog Viewer provides the designer with a graphical display of the design tree representing the different design versions of the database. Figure 6.1 shows an example of the Catalog Viewer. The root of the design tree represents the first version or catalog of the database design. A child of a catalog is created by the **COPY** operation in the Current Catalog area. The designer decides when a particular design version should be saved. A particular catalog is activated by selecting **VIEW**.

The *Relation Viewer* (see Figure 6.2) is used to view individual relations in a design version. It displays the name of the relation, its attributes, functional dependencies, keys, and an example table. Editors that can add, delete, or modify the attributes or functional dependencies can be invoked from this screen.

The *Decomposition Viewer* (see Figure 6.3) is activated through the Relation Viewer. This screen displays a scheme obtained by decomposing the relation into either 3NF or BCNF, depending on the selection done in the Relation Viewer. The Decomposition Viewer also shows properties of the decomposition, such as normal form, preservation of dependencies, and lossless join. The attributes of each relation in the decomposition, together with the relations example table are also displayed. The keys or functional dependencies projected onto each relation in the decomposition can be displayed by selecting the appropriate **Keys** or **FDs** option. Many alternative decompositions may exist for a given relation. The designer can examine alternative schemes by selecting the NEXT SCHEME option.

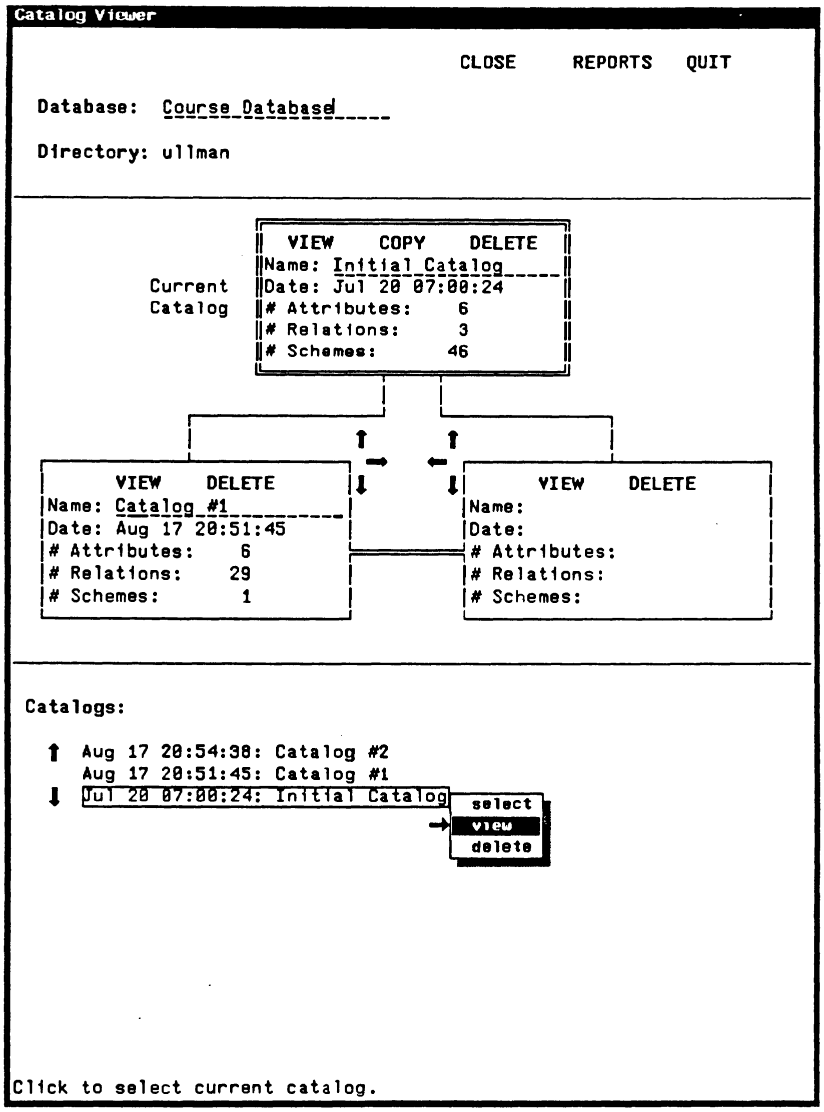


Figure 6.1. The Catalog Viewer

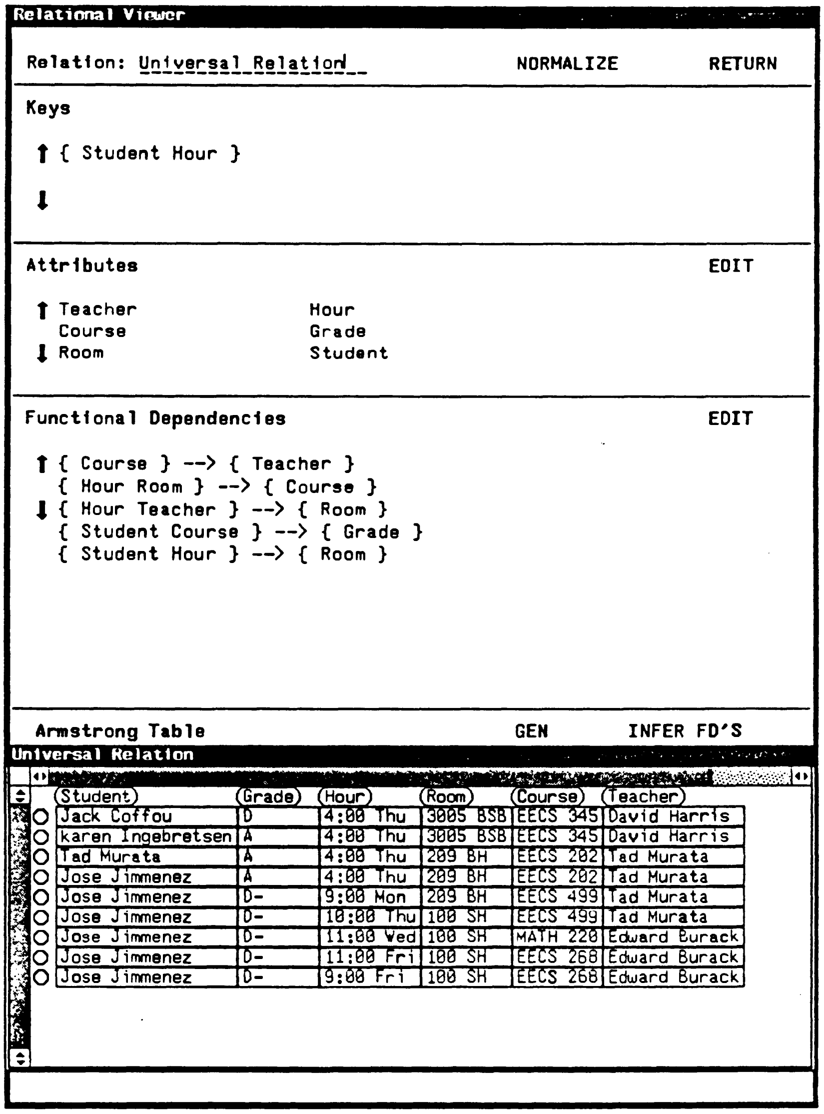


Figure 6.2. The Relation Viewer

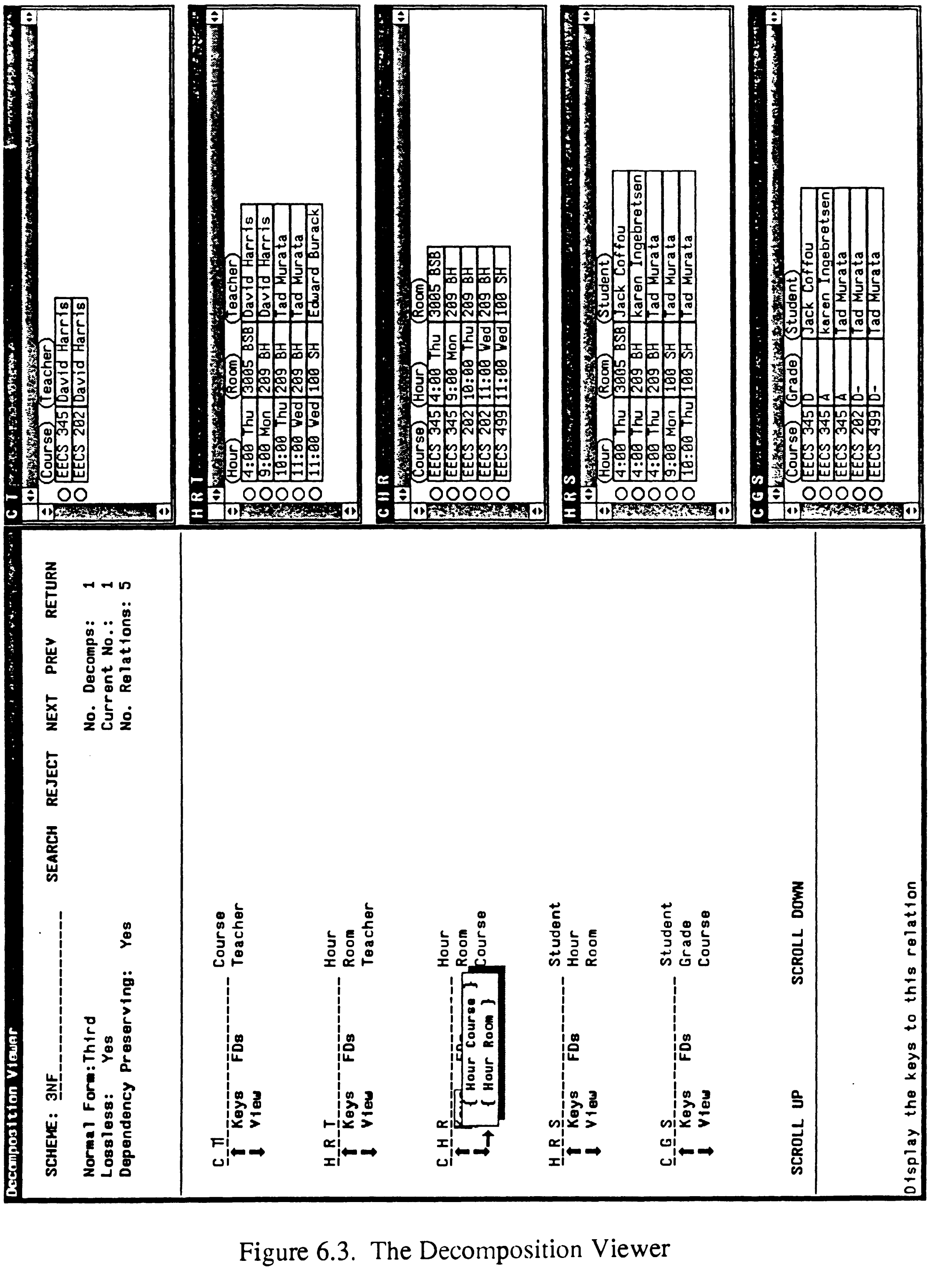


Figure 6.3. The Decomposition Viewer

The *Attribute Editor* is the screen where the database designer specifies and updates attributes, their formats and their domain types. The Attribute Editor can be used as a Data Dictionary to store the description and use of an attribute. It is also possible to list the relations in the database design containing a particular attribute. The domain type of an attribute is defined in the *Domain Editor*. Here, the range and type of an attribute is specified. The values are used in the example tables to create a realistic example of a relation instance.

The functional dependencies and keys are specified and updated through the *Functional Dependency Editor* (see Figure 6.4). A number of specialized features are associated with the editor. The editor will check consistency of any new dependency before adding it to the current set. Keys are dynamically found and displayed, and the set of dependencies can be reduced to a more compact cover by selecting the **COVER** menu. Two different types of covers are generated: a *minimal cover* and a *canonical cover*.

The Functional Dependency Editor also provides an example relation that is dynamically updated to be an Armstrong relation for the functional dependencies displayed. The database designer can reorganize and modify the example table. Interchanging rows or columns, sorting, and adding and deleting rows or columns are some of the operations allowed on an example table. The functional dependencies that are satisfied by the example table can be inferred by selecting **INFER**.

DBE has a number of other functions and editors, such as a *Query Editor*, that can ease the design of relational databases and help the designer in selecting an appropriate design. For further information on the interface and use of the tool, we refer to [Mill88].

6.3. The DBE Internal Structures

A first prototype for *DBE* was implemented in the programming language LISP on a Symbolics machine. There were two reasons for choosing that environment. The first was to facilitate quick prototyping. LISP provides data structures and functions that make the implementation of traditional design algorithms straightforward. In particular, lists and their associated operations are a natural representation for functional dependencies. The second reason was that the graphics software available on the Symbolics was easy to use and provided support for displaying and manipulating example tables, along with other windows containing current information needed in the design process.

However, as we experimented with this prototype, we soon realized that a practical design tool could not be implemented without using special data structures and functions that would support an efficient implementation of the relational design algorithms. Even for very small databases, the amount of main memory used by the design tool was prohibitive, and the algorithms for generating Armstrong relations and inferring functional dependencies were extremely slow.

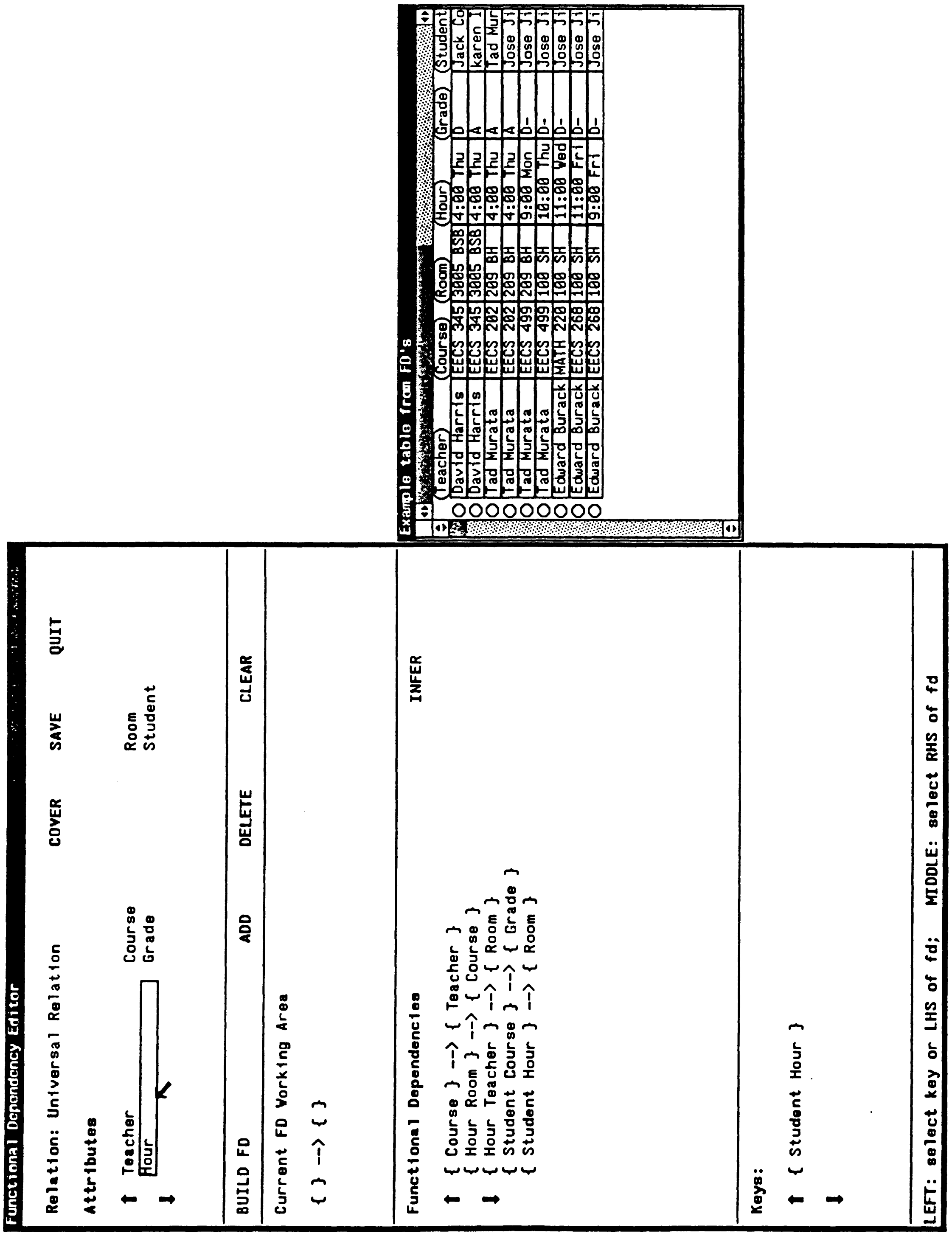


Figure 6.4. The Functional Dependency Viewer

Thus, in our second implementation of *DBE*, we opted for a more flexible programming environment and turned our attention to system efficiency. The current version of *DBE* is written in the C programming language and runs under UNIX on SUN/3 workstations. It consists of 59,000 lines of code and runs comfortably with 4 megabytes of main memory. The graphics interface uses the Suntools package. Below, we briefly describe and motivate the memory management scheme and the main data structures in the current implementation. For a complete description of the data structures, we refer to [Mill88].

6.4. The DBE Memory Management

For efficiency reasons, we do not use use the complete memory management routines provided by UNIX, but allocate large blocks of memory at a time. Once this memory is allocated, we maintain our own free lists for the main data structures required by the design algorithms to control locality of reference and paging. To help minimize wasted memory, we store all character strings, stripped of trailing and leading blanks, exactly once in a hash table and return a list element pointer from the appropriate bucket. A consequence of this storage scheme is that string equality reduces to pointer equality. This is a motif throughout the design of the system.

6.4.1. Attributes and Domains

An attribute consists of a character string name and a domain. It, too, is stored only once, and all occurrences of the attribute (e.g., as part of a relation scheme or a functional dependency) are represented by a pointer to the appropriate structure. Each attribute belongs to a domain, which gives *DBE* information about what values it can take on. For each domain, there is a function that returns a list of valid values that is used in generating the Armstrong tables. The two ways we have implemented this function are by returning elements from a linked list of values that have been stored in a domain database (e.g. "cities," "names," and other finite domains) and by computing the next element from a seed (e.g., "integers," "money," or other infinite domains).

6.4.2. Tables

A table, or relation instance, is stored in a compact structure that is designed to facilitate insertion, deletion, modification, or copying of a tuple. Since *DBE* uses example tables to represent functional dependencies, it is important to support these functions efficiently. Our table data structure is similar to one used in the main memory database system *OBE* [Whan86].

As shown in Figure 6.5, tables consist of a tuple area and an array of offsets to tuples. A tuple is an array of pointers to character strings. A table also has a descriptor, which contains information such as the current number of tuples in the table, and a permutation vector used to display the table columns in different orders.

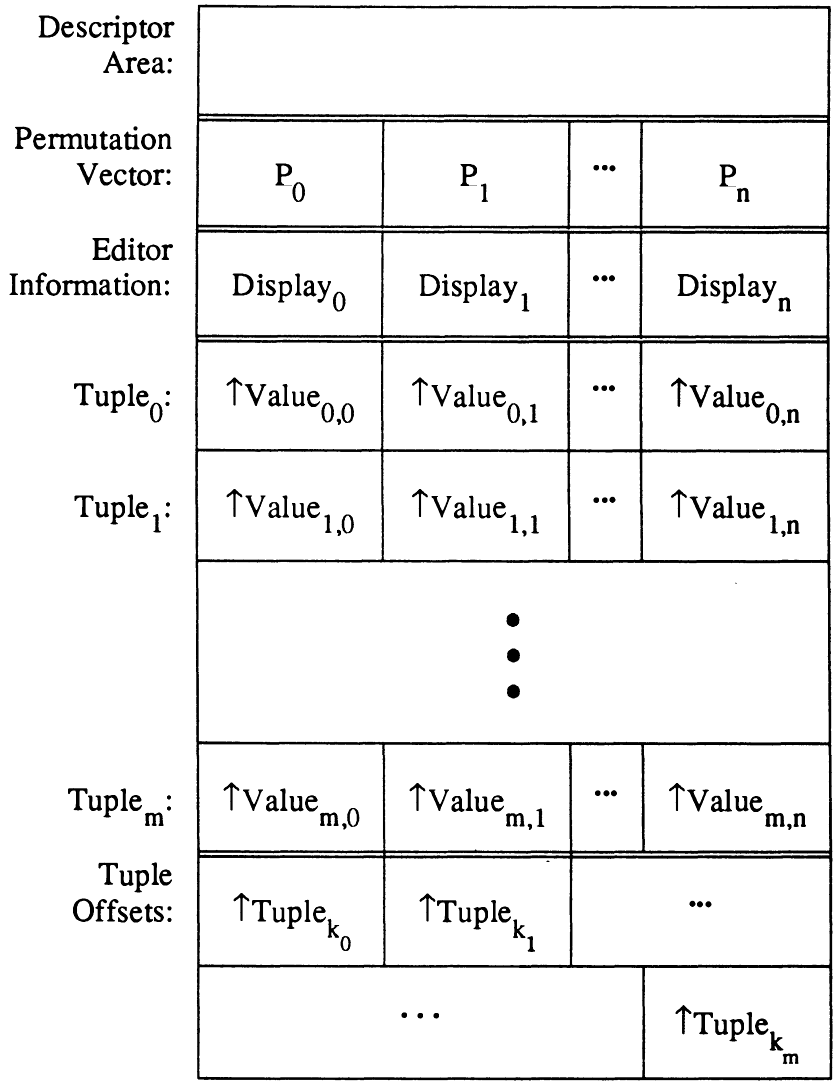


Figure 6.5. The *DBE* Table Structure

6.4.3. Attribute and Dependency Sets

Relational design algorithms operate on sets of attributes, that represent relation schemes or left- or right-hand sides of functional dependencies. Thus, it is important to efficiently support set operations such as union, intersection, subset, etc. In *DBE*, attribute sets are internally represented as binary vectors. The attributes in the database are ordered, $R = A_0, A_1, \cdots, A_{n-1}$. A set $S \subseteq R$ is represented as a binary number $(i_0, i_1, ..., i_{n-1})$, with i_j=1 if A_j is in S, and i_j=0 otherwise.

With this data structure, we are able to support an efficient implementation of set operations (union, intersection, difference), boolean operations (set-membership, subset, proper subset), and functions to add or remove an attribute from a set. For most practical databases, these operations are accomplished in few machine instructions. These functions constitute a basic layer of *DBE* and are invoked by the higher level functions.

6.5. Algorithms for Dependency Inference

The problem of inferring all the dependencies satisfied by a relation r is intrinsically exponential in the number of attributes in r. A naive approach to the problem consists of checking for all subsets $X \subseteq R$ and all attributes $A \in R-X$ whether r satisfies the dependency $X \rightarrow A$. Testing whether the dependency $X \rightarrow A$ holds in r requires inspecting all pairs of tuples and checking that all tuple pairs that agree on X also agree on A. This approach yields an exponential algorithm, since $n2^{n-1}$ possible dependencies of the form $X \rightarrow A$ must be tested. Checking each dependency also requires $p(p-1)$ steps, where p is the number of tuples. Thus this naive method is not practical. It is inefficient and produces many redundant dependencies.

Below, we describe two algorithms for dependency inference that have recently been proposed by Mannila and Räihä [Mann85, Mann87] and propose a number of optimizations to one of these algorithms.

6.5.1. Algorithm 1

To reduce the number of dependencies and the time to find them, we must exploit global information in the relation instance. Mannila and Räihä [Mann87] propose the following approach for constructing the set of determinants for an attribute A. Define

$agr(s,t) = \{B \in R \mid s[B] = t[B],\ s,t \in r\}$ and
$disagr(s,t) = \{B \in R \mid s[B] \neq t[B],\ s,t \in r\}$

and group together those *disagr(s,t)* that disagree on attribute A:

$$det(A) = \{disagr(s,t)-A\} \mid s,t \in r,\ A \in disagr(s,t)\}$$

We know that a dependency $X \rightarrow A$ holds in r, the current value of R, if all pairs of tuples that disagree on A also disagree on X. Thus, all attribute sets X that can determine A must contain at least one element of each set in *det(A)*. In other words, *lhs(A)*, the set of all determinants of A, can be constructed as:

$$lhs(A) = \{\ X \subseteq R - A,\ X \cap Y \neq \emptyset \text{ for all } Y \in det(A)\}$$

In order to avoid inferring redundant and transitive dependencies, the sets *det(A)* should be reduced before proceeding with the construction of *lhs(A)*. Redundant dependencies can be avoided by removing supersets in *det(A)*. Transitive dependencies can be avoided by comparing *det(A)* with the determinants of other attributes. To illustrate the algorithm let us consider the following Customer relation from Figure 6.6.

	name	**addr**	**ship-date**	**item-code**
t_0	Terry	Minneapolis	7-17-86	st
t_1	Bill	Chicago	7-17-86	le
t_2	Paul	Chicago	7-17-86	sd
t_3	Paul	Chicago	8-20-87	xy
t_4	Paul	Chicago	8-20-87	px

Figure 6.6. Customer Relation

By inspecting *disagr(s,t)*, we find that

det(name) = { {address,item-code},
{address,ship-date,item-code},
{ship-date,item-code}, {item-code} }
det(addr) = { {name,ship-date,item-code},
{name,item-code} }.

By removing supersets, we reduce these determinants to

det(name) = { {item-code} }
det(addr) = { {name, item-code} }

To avoid producing the transitive dependency *item-code→addr*, the attribute item-code should be removed from *det*(addr).

The complete version of Algorithm 1, adapted from [Mann87], is given in Algorithm 6.1. In general, Algorithm 1 is exponential in the number of attributes.

6.5.2. Algorithm 2

Another approach, proposed in [Mann85], is an iterative construction of *lhs(A)*. Each iteration compares a pair of tuples *s,t*, for which it constructs an agree set *agr(s,t)*. When the tuples disagree on an attribute *A* but agree on sets that are currently in *lhs(A)*, these sets are augmented by adding attributes that disagree. The algorithm is sketched below.

Algorithm 6.1. Algorithm 1

Input: A relation instance r over relation scheme R.

Output: A set F of functional dependencies over R such that r satisfies a dependency $X \rightarrow A$, if and only if $X \rightarrow A \in F^{+}$.

```
1. {Iterate over all pairs of tuples in r}
      for all s,t ∈ r, s ≠ t do
            compute D:=disagr(s,t);
            if D ∈ disagr then
                  disagr := disagr ∪ D;
                  for all A ∈ D do
                        det(A) := det(A) ∪ D − {A};
                        <remove any supersets from det(A)>;

2. {Transitivity reduction: remove redundant attributes from det(A).}

3. {Infer functional dependencies from the det(A) sets}
      for all A ∈ R do lhs(A):=∅; search_lhs(1, det(A), lhs(A), ∅);

4. {Construct F, initially F=∅}
      for all A ∈ R do
            for all X ∈ lhs(A) do F := F ∪{ X → A }

5. return(F);

procedure search_lhs(i, S, lhs, cur_lhs);
{find the sets that determine A given det(A) = S= {S1, S2, . . . , Sk}}
      if i = | S| then {Base Case}
            if Si ⊆ lhs then lhs := lhs ∪ cur_lhs
            else for all C ∈ Si and C ∉ cur_lhs do
                  lhs := lhs ∪ (cur_lhs ∪ C)
      else {Recursive Case}
            if Si ⊆ lhs then search_lhs(i+1, S, lhs, cur_lhs)
            else for all C ∈ Si and C ∉ cur_lhs do
                  cur_lhs := cur_lhs ∪ C;
                  search_lhs(i+1, S, lhs, cur_lhs);
                  cur_lhs := cur_lhs − C
```

```
1. {Initialize}
   for all A ∈ R let lhs(A) := {∅};

2. {Iterate over all pairs of tuples in R}
   for all t, s ∈ r, t ≠ s,  do
        for all A ∉ agr(s,t)
        {update lhs(A) with respect to t and s}
              for all X ∈ lhs(A) and X⊆agr(s,t) do
                    lhs(A) := lhs(A) – {X};
                    for all B  ∉ agr(s,t) do
                          {correct the determinants}
                          lhs(A) := lhs(A) ∪ {XB};
                    {remove supersets from lhs(A) }
                    while lhs(A) contains X and Y such
                          that X ⊆ Y  and X ≠ Y do
                          lhs(A) := lhs(A) – Y
```

A closer look at this algorithm reveals that it can be improved by:

1. eliminating trivial dependencies in early iterations.
2. using unique agree sets, instead of considering all pairs of tuples.
3. reducing the number of agree sets by eliminating sets that can be obtained by intersection.

(1.) Avoiding trivial dependencies.
The algorithm can be improved by detecting early attributes without a left-hand side and by not producing sets containing A when updating *lhs(A)*. If after an iteration *lhs(A)* = $\{\{A\}\}$, there is no need to try updating *lhs(A)* anymore. Since this condition occurs when two tuples agree on all attributes but A, it indicates that no non-trivial dependency exists with A as a left-hand side.

(2.) Unique agree sets.
For most relations r the number of distinct *agr(s,t)* is less than $p(p-1)/2$, where p is the number of tuples. Consider again our Customer relation in Figure 6.6. Notice that tuple pair (t_1, t_3) has the same agree set as (t_1, t_4) (i.e. {addr}). These tuple pairs will produce exactly the same modifications to the left-hand sides. Since it is inefficient and unnecessary to consider the same agree set twice, we sort the agree sets to remove duplicates. We are also updating the left-hand sides with sets *agr(s,t)* ordered in decreasing cardinality. The conjecture here is that the left hand sides will grow more slowly this way.

(3.) Reduced agree sets.
It is possible to reduce the number of sets from *agr(r)* handled by removing sets that are exactly the intersection of two other sets of *agr(r)*. In our example in Figure 6.6, we have the following sets agr_1={addr ship-date}, agr_2={name addr}, agr_3={addr}, all members of *agr(r)*. It turns out that $agr_3 = agr_1 \cap agr_2$ can be removed from *agr(r)* without changing the dependencies inferred since the updates to the left-hand side caused by agr_3 will also be caused by agr_1 and agr_2.

Theorem 6.1: If $agr_1, agr_2, \ldots, agr_k, agr_p \in agr(r)$ and $agr_p = agr_1 \cap agr_2 \cap \cdots \cap agr_k$, then $agr(r) - \{agr_p\}$ infers the same dependencies as *agr(r)*.

Proof: It is known that given a dependency set *F*, for any relations *r* that satisfies *F*, the set *agr(r)* contains the minimal subfamily of generators *GEN(F)* and that every element of *agr(r)* is F-closed [Beer84b]. Thus, *GEN(F)* is the smallest *agr(r)* that a relation satisfying *F* can have.

By definition of *GEN(F)*, all F-closed sets can be obtained by intersecting members of *GEN(F)*. If $agr_p \in agr(r)-GEN(F)$ and $agr_1, agr_2, \ldots, agr_k \in GEN(F)$,then the theorem is clearly true. If some $agr_i \in agr(r)-GEN(F)$ where $i= 1,\ldots,k$, the theorem still holds since agr_i can be substituted by a conjunction of elements from *GEN(F)*. □

It is interesting to draw an analogy between this theorem and the finding of small Armstrong relations (see Chapter 3).

In the optimization we propose, only elements that are the intersection of two elements are removed. Removing elements that are intersections of three or more elements is time consuming and does not provide a significant reduction of *agr(r)*. The complete version of Algorithm 2, in its optimized version, is given in Algorithm 6.2.

6.6. Performance Measurements

In general, the inference algorithms described in the previous section are exponential in time and space. It is difficult to forecast the execution time for a dependency inference algorithm, since the behavior of the algorithm depends on the output it produces (that is, the dependencies) as well as on the number of attributes and cardinality of the input relation. Furthermore, we found that implementation (data structures for the domains, relations, and dependencies, and memory management) had a dramatic impact on the performance of the algorithms.

In a previous prototype of the *DBE* system, implemented in LISP on a Symbolics machine, the inference algorithm was extremely slow and we were wondering whether it would be feasible to keep it as a component in a design tool. Typically, it took 15 minutes to infer dependencies from small examples (e.g., 20 tuples, 10 attributes) on a Symbolics 3640. With the new and refined implementation of *DBE*, Algorithm 6.2. Algorithm 2, Optimized

Input: A relation instance r over relation scheme R.

Output: A set F of functional dependencies over R such that r satisfies a dependency $X \to A$, if and only if $X \to A \in F^+$.

1. {Initialize}
    for all $A \in R$ **do** $lhs(A) := \{\emptyset\}$;
    $agr := \{\emptyset\}$;

2. {find *agr* set for r}
    for all $t, s \in r, t \neq s$ **do begin**
        Y := $agr(t,s)$; { Y = attributes on which t and s agree.}
        if Y $\notin$ agr **then** $agr := agr \cup$ Y

3. { find reduced agree set (r_agr) for R.}
    for all $a, b \in agr, a \neq b$ **do**
        if ($d := a \cap b) \in agr$ **then** $agr := agr - d$;
    $r\_agr := agr$;

4. {perform *inference* on r_agr}
    for all $g \in r\_agr$
        for all $A \notin g$ **do**
            for all $X \in lhs(A)$ **and** $X \subseteq g$ **do**
                $lhs(A) := lhs(A) - X;$
                for all $D_i \notin$ g **do**
                    $lhs(A) := lhs(A) \cup XD_i$
        while $lhs(A)$ contains X **and** Y **and** $X \subseteq Y$ **do**
            $lhs(A) := lhs(A) - Y$;
        while $lhs(A)$ contains set X **and** $A \in X$ **do**
            $lhs(A) := lhs(A) - X$;

where more attention was paid to designing efficient data structures, managing memory, and optimizing the algorithms, it became possible to obtain more meaningful performance measurements for the inference process.

To get a better insight on the performance of the algorithms and on the effectiveness of the optimizations that we proposed, we experimented with running the two inference algorithms and their optimized versions on different example relations. In choosing these relations, we varied the following parameters:

1. number of attributes,
2. number of tuples,
3. number of dependencies, and
4. degree of normalization.

For each test, we recorded the CPU time, memory usage, and the size of the output produced (i.e., the number of dependencies). The two algorithms and the optimized version of Algorithm 1 usually inferred different but equivalent sets of dependencies. To normalize the time measurements with respect to the number of dependencies produced, we added the time required for finding a minimal cover.

6.6.1. Experiment 1: Feasibility

The goal of our first experiment was to obtain an order of magnitude for the time to run the inference algorithms, on arbitrary example relations. For this experiment, we used 6 relations, varying the number of tuple attributes from 7 to 50 and the number of tuples from 7 to 245. The number of dependencies was also varied, from 8 to 57, but kept close to the number of attributes in each example. The attributes and the dependencies were chosen to model a "realistic" Customer Relation. (Additional attributes to those shown in Figure 6.6 were added for the larger relations). The actual data was produced using an algorithm to generate Armstrong relations [Beer84, Mann85].

The CPU time, in seconds, required for inferring the dependencies satisfied by our six example relations using the optimized version Algorithm 2 is shown in Table 6.1. They range from 0.02 seconds for a very small relation (Rel1, 7 attributes and 7 tuples), to 15 minutes for a larger relation (Rel6, 60 attributes and 236 tuples). This experiment shows that in spite of the exponential complexity of the problem, the time for inferring dependencies satisfied by a random example relation is not prohibitive. For small relations, up to 17 tuples and 33 tuples, the algorithm runs in less than one second. For Rel4, which had 21 attributes and 113 tuples, the time was under 10 seconds.

Table 6.1. Random Relations
Inference Time in Seconds

Relation	Atrs	Tpls	Fds*	Inference Time **
Rel1	7	7	8	0.02
Rel2	17	20	13	0.36
Rel3	17	33	20	0.83
Rel4	21	113	17	9.50
Rel5	50	245	57	732.28
Rel6	60	236	56	993.58

* Number of dependencies in a minimal cover.
** Time is for Algorithm 2, optimized.

6.6.2. Experiment 2: Normalized Relations

At this point, it is not known whether any of the optimizations proposed for Algorithm 2 reduces the complexity of the algorithm in this particular case. To see how the two algorithms would compare, we constructed BCNF relations with a unique composite key and different cardinalities. The inference time with both algorithms is shown in Table 6.2. We were surprised to find out that the algorithms had similar performance. Algorithm 2 only became slightly faster as we increased the number of attributes above 40 in the input relation.

Even then, it seems that this time efficiency was at the expense of space. We kept track of the memory allocated by *DBE* during execution of the two algorithms. Some of this data is shown in Table 6.3, for three normalized relations, with 20, 80, and 180 tuples. As a baseline for comparison with the storage required for the input relation, we have included the memory that the *DBE* storage subsystem requires for the table itself and the character strings. Algorithm 1 required twice or more the amount of memory than Algorithm 2. The space seems to be needed to store all the *det(A)* sets.

Table 6.2. BCNF Relations
Inference Time in Seconds for Algorithms 1 and 2

Relation	Atrs	Tpls	Fds	Alg1	Alg2
NRel1	10	11	9	0.06	0.6
NRel2	15	16	14	0.24	0.14
NRel3	20	21	19	0.44	0.36
NRel4	25	26	24	0.78	0.72
NRel5	30	31	29	1.34	1.18
NRel6	35	36	34	2.24	2.20
NRel7	40	41	39	3.18	3.30
NRel8	45	46	44	4.52	4.86
NRel9	50	51	49	6.02	6.74

Table 6.3. BCNF Relations
Memory Requirements in KBytes for Algorithms 1 and 2

Relation	Tpls	Table	Strings	Alg1	Alg2
NRel3	20	2KB	.8KB	11KB	6KB
NRela	80	8KB	1.1KB	115KB	35KB
NRelb	180	17KB	3.2KB	295KB	151KB

6.6.3. Experiment 3: Unnormalized Relations

In our next experiment, we considered highly unnormalized relations, which have been used in the literature to show the exponential complexity of other problems in database design [BDF84]. The relations satisfy one of the following dependency sets:

1. $F = \{A_i \rightarrow A_{i+1} \mid i = 1,3,\ldots 2m-1\} \cup \{\bigcup_{j\ even} A_j \rightarrow B\}$.
2. $G = \{A_i A_j \rightarrow B \mid 1 \le i < j \le 2m\}$.

We generated three relations satisfying the F type of dependencies (with 4, 6, and 8 Fds), and four relations satisfying the G type of dependencies (with 6, 45, 91, and 190

Fds). The times for Algorithms 1 and 2, in their optimized version, are shown in Table 6.4. This experiment clearly shows the exponential increase in inference time for both algorithms, as the number of dependencies increases (linearly for the F relations, and as the square of the number of attributes for the G relations). Another observation from Table 6.4 is that, as the number of dependencies increases, Algorithm 1 performs better than Algorithm 2.

Table 6.4. Highly Unnormalized Relations
Time in Seconds for Algorithms 1 and 2

Relation	Atrs	Fds	Tpls	Alg1-Opt	Alg2-Opt
RelF_7	7	4	5	0.02	0.02
RelF_11	11	6	6	0.12	0.24
RelF_15	15	8	22	0.24	4.34
RelG_8	8	6	12	0.02	0.04
RelG_14	14	45	24	0.32	0.46
RelG_18	18	91	32	0.78	1.66
RelG_24	24	190	44	2.44	6.74

6.6.4. Experiment 4: Optimization of Algorithm 2

This experiment was designed to measure the effectiveness of the optimizations for Algorithm 2 that were described above. In Table 6.5, we show the times for the algorithm and for its optimized version, on a cross section of relations, including random relations (Rel1 to Rel6), two normalized relations, and two highly unnormalized relations (FRel3 and GRel3). The percent of improvement ranges from 43% to 98%. The optimization is less effective when the number of dependencies is small (Rel1 and FRel3), but it seems to be equally effective for normalized and unnormalized relations. We were surprised that simple heuristics would improve so dramatically the performance of the algorithm. From other measurements (not shown here), it seems that most of the improvement comes from looking at unique agree sets, instead of tuple pairs, and in some cases, from reducing the agree sets.

Removal of the supersets in the temporary *lhs(A)* consumed a significant proportion of the overall time for the inference. In another test, we measured the impact of removing supersets. If supersets were not removed at the end of each iteration, the run time of the algorithm was dramatically higher and the memory requirements increased substantially.

Table 6.5. Optimization of Algorithm 2
Inference Time in Seconds and Percent Improvement

Relation	Atrs	Tpls	Fds	Alg2	Alg2-Opt	Impr
Rel1	7	7	8	0.10	0.02	80%
Rel2	17	20	13	1.58	0.36	77%
Rel3	17	33	20	27.98	0.84	97%
Rel4	21	113	17	273.44	9.50	96%
Rel6	60	236	57	57,613.70	993.58	98%
NRela	21	82	15	54.24	9.70	82%
NRelb	21	184	18	307.38	16.26	94%
FRel3	15	22	8	7.68	4.34	43%
GRel3	15	32	91	135.46	1.66	98%

6.7. Summary

We have addressed a number of feasibility and performance issues in automatic database design by describing the DBE prototype. The DBE tool makes important knowledge in relational design theory automatically and transparently available to the database designer. It can assist the database designer in the specification of a relational database through graphically displayed example tables and queries. It has an underlying storage subsystem for attributes, relations, and dependencies that supports a time and space efficient implementation of design algorithms.

Inferring the functional dependencies satisfied in an example relation is one of the key operations in Design By Example. A naive approach to dependency inference is not practical because it would be unacceptably slow in an interactive design session. We have investigated the feasibility of inferring dependencies from a given relation by optimizing and implementing two recently proposed algorithms [Mann85, Mann87],

We have presented an extensive set of experiments with the dependency inference algorithms, where we varied the cardinality, number of attributes, number of dependencies, and degree of normalization of the input relations. The experiments were performed with the current version of *DBE*, on a SUN/3 workstation with four megabytes of main memory. For practical examples (10 to 100 tuples, 10 to 20 attributes), our tests showed that dependencies can be inferred in under 10 seconds. For small examples (with about 20 tuples and 20 attributes), dependencies were inferred in less than one second, even in the case of highly unnormalized relations with a large number of dependencies. These performance results indicate that it is practical to include a dependency inference function in an automatic database design tool.

CHAPTER 7

Summary and Future Work

In this dissertation, we presented methods and algorithms for automating the design of relational databases. Usually, database design is done manually by a skilled expert. The design process is often difficult, time consuming, and error-prone. Automation, therefore, can be of great help. We proposed an expert design tool for relational databases, using the pure relational model with functional and multivalued dependencies.

We have focused on the relational model, rather than higher level data models. In order to motivate the use of this model, we first surveyed the Entity-Relationship and the Iris models. In this survey, we showed that semantic data models provide more expressive power, but that the design theory for these models is currently incomplete and does not provide a foundation as well developed as the relational design theory.

In relational design theory, the most commonly used relationship is the functional dependency. We summarized earlier work on functional dependencies and the design algorithms that manipulate them. We presented a new way of representing functional dependencies, as a semilattice and as an attribute-fd graph. The attribute-fd graph provides a unified way for various design algorithms to manipulate functional dependencies. The algorithms for finding the closure and the projection of functional dependencies and finding keys can be viewed as traversing the attribute-fd graph. The attribute-fd graph improved the time complexity of some of these algorithms. Through the semilattice representation, we defined the basis $B(F)$, and showed how it can be used to find multiple BCNF decompositions and the generator of the F-closed sets. Previous research on normalization algorithms does not emphasize the existence of multiple decompositions. From a theoretical point of view, finding one decomposition with desirable properties is satisfying. However, from a practical point of view it is desirable to consider multiple decompositions that have the same characteristics (e.g., BCNF, lossless). Certain decompositions seem more natural to the database designer, making queries easier to write.

Our algorithm for finding *multiple* BCNF decompositions based on the basis $B(F)$ generalizes the Tsou-Fisher algorithm, which finds *one* BCNF decomposition. We would like to improve the Tsou-Fisher algorithm so that it will produce schemes with fewer relations, thereby avoiding unnecessary steps such as decomposing relations that already are in BCNF. It would also be of interest to generate decompositions in an order such that the lossless and dependency preserving decompositions are found first, if they exist. The complexity of the presented algorithm could be reduced somewhat by reducing the candidate schemes to those that satisfy the above mentioned properties.

Finding multiple 3NF decompositions is planned for future research. An approach similar to finding multiple BCNF decompositions using our concept of a basis should be studied further. It is likely that this will be successful since the basis contains all the dependencies that can occur in a canonical cover. Finding different 3NF decompositions is very similar to finding different canonical covers for a functional dependency set. Also, in this case, it is important to find a lossless and dependency preserving decomposition first, which is known to always exist for a set of functional dependencies. This makes it possible to automatically find and display the schemes that the database designer will probably select.

Restructuring an existing database is another desirable use of an automatic database design tool. In this context, one would like to infer the dependencies that hold for the relational database scheme. A scheme has internal and external dependencies. The internal dependencies hold within a relation, while the external hold between the relations. We presented an algorithm to infer the external multivalued dependencies from an acyclic relation scheme. For the inference of functional dependencies within a relation, we optimized previously proposed algorithms. We also proposed to use these algorithms to infer functional dependencies from example tables containing application data provided by the database designer. An inference algorithm in conjunction with the algorithm to generate an example table that exactly satisfies the functional dependencies gives a foundation for the database designer to verify that the dependencies are correct. Generation of example tables that could be an instance of a relation in an actual database enables the application specialist to work with a familiar view. This interface would make it easier to capture errors than simply displaying a set of functional dependencies.

Further studies of inferring external multivalued dependencies from cyclic relation schemes would be appropriate since designs are often acyclic. This problem is harder to solve, since separators in a cyclic hypergraph will no longer correspond to articulation sets only, but can be contained in disjoint hyperedges.

Because queries often impose implicit constraints to the natural grouping of attributes into a relation scheme, we proposed to integrate query design into the logical design phase. The candidate schemes can also be rated according to how queries will perform against them. We defined queries in a universal relation interface where queries are specified by attributes in the database, independent of a particular design. We presented an algorithm for finding the exact query formulation for a particular candidate design. The algorithm finds multiple ways of accessing the data in the case that the query specified is ambiguous. An ambiguity in the query can be a result of an erroneous specification, a poor semantic specification for the database application, or simply an indication that the current design is not natural. For some queries, our method does not find solutions because lossy paths or cycles are not allowed to occur in the set of relations used to access the data. The lack of a join path for a query indicates an anomaly with either the candidate scheme or the query itself. Avoiding cycles by restricting subspanning trees to be acyclic hypergraphs seems to be too restrictive. Improving the algorithm to find subspanning trees using other definitions of cyclicity can result in finding solutions to acceptable queries that otherwise would not have been found.

Automatic query formulation is not possible for yet another type of query. Queries that require a relation to be joined with itself cannot be found by our join path finding algorithm or by any of the other join path finding algorithms surveyed in this dissertation. A relation occurs only once in the database; thus the algorithms will find only one occurrence of a relation for each query formulation.

We briefly discussed ways in which the relation scheme can be modified in order to produce a candidate design that is more suited for the queries. An appropriate interface for modifying the scheme and inspecting the query formulations remains to be found. One approach would be to use Armstrong relations to produce synthetic relations for the database and perform the query on this database. Using example relations makes it possible to display dangling tuples and show whether the different query formulations produce different answers. This can enhance the possibility of uncovering anomalies in the database design and help the designer to better understand the database and its queries.

Inclusion dependencies should be added to the set of dependency types used in the design tool. This type of dependency occurs frequently in applications and would be a useful feature of the design tool. Unfortunately, the theory of inclusion dependencies is not as extensive as the theory of functional and multivalued dependencies. Further investigation is needed to incorporate inclusion dependencies in the design tool.

We investigated the feasibility of inferring functional dependencies from an example relation. The complexity of the inference problem is inherently exponential. However, the algorithms performed well when the input relations had certain characteristics. An extensive set of experiments were presented, where dependencies were inferred from example relations with different cardinalities, number of attributes, and degree of normalization. We concluded that, for practical example relations, an adequate implementation of a dependency inference function leads to acceptable interactive response time. It would be interesting to study the performance of other relational design algorithms such as the projection algorithm and the exponential BCNF decomposition algorithm. The complexity of the algorithms are based on the functional dependency sets, and, for most applications, these sets can have less extreme effects on the algorithms than their complexities indicate. Performance testing can validate this.

In summary, the main contribution of this dissertation is a new approach for efficiently representing and manipulating functional and multivalued dependencies. For functional dependencies, we proposed the attribute-fd graph representation of a functional dependency set F and the basis $B(F)$, a compact representation of F^+. The attribute-fd graph provides a suitable representation for the classical design algorithms and the basis $B(F)$ made it possible to compute multiple BCNF decompositions and generate small Armstrong relations. We also presented algorithms for inferring multivalued dependencies from acyclic relation schemes and optimized existing algorithms for inferring functional dependencies from a relation instance. We introduced a new approach to logical design by using queries to discover anomalies in the candidate design. These contributions provide a sound foundation for further research in methods to automate relational database design. Some of the future work would be to find methods for computing multiple 3NF decompositions, to generalize the algorithm for inferring multivalued dependencies form a relation scheme, and to generalize the algorithm for finding join paths for queries specified over a universal relation interface. Further performance and feasibility studies can give an indication of how practical complex design algorithms are and whether they can be used to fully automate the database design process.

List of References

[Aho79a] Aho A.V., Beeri C., and Ullman J.D., "The Theory of Joins in Relational Databases," *ACM Transactions on Database Systems*, 4:3,1979.

[Aho79b] Aho A.V., Sagiv C., and Ullman J.D., "Efficient Optimization of a Class of Relational Expressions," *ACM Transactions on Database Systems*, 4:4,1979.

[Aho79c] Aho A.V., Sagiv C., and Ullman J.D., "Equivalences Among Relational Expressions," *SIAM J. on Computing*, 8:2,1979.

[Alba85] Albano A., Occhiuto M.E., and Orsini R., "Galileo: Strongly-typed, Interactive Conceptual Language," *ACM Transactions on Database Systems*, 10:2,1985.

[Arms74] Armstrong W.W., "Dependency Structures of Data Base Relationships," *Information Processing 74*, 1974.

[Aror84] Arora A.K. and Carlson C.R., "Normalization Could be Useful," *The Computer Journal*, 27:1,1984.

[Ausi83] Ausiello G., D'Atri A., and Sacca D., "Graph Algorithms for Functional Dependency Manipulation," *J. ACM*, 30:4, 1983.

[Bati79] Batini C. and D'Atri A., "Rewriting systems as a tool for relational data base design," *Proc. on Graph-Grammars and Their application to Computer Science*, Lecture Notes in Computer Science 73, Springer-Verlag, New York, 1979.

[Beec83] Beech D. and Feldman J.S., "The Integrated Data Model: A Database Perspective," *Proc. Ninth Int. Conf. on Very Large Data Bases*, 1983.

[Beer77] Beeri C., Fagin R., and Howard J.H., ''A Complete Axiomatization for Functional and Multivalued Dependencies in Database Relations,'' *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1977.

[Beer78] Beeri C., Bernstein P.A., and Goodman N., ''A Sophisticate's Introduction to Database Normalization Theory,'' *Proc. Fourth Int. Conf. on Very Large Data Bases*, 1978.

[Beer79] Beeri C. and Bernstein P.A., ''Computational Problems Related to the Design of Normal Form Relational Schemas,'' *ACM Transactions on Database Systems*, 4:1,1979.

[Beer80] Beeri C., ''On the Membership Problem for Functional and Multivalued Dependencies in Relational Databases,'' *ACM Transactions on Database Systems*, 5:3,1980.

[Beer81a] Beeri C., Fagin R., Maier D., Mendelzon A., Ullman J., and Yannakakis M., ''Properties of Acyclic Database Schemes,'' *STOC*, 1981.

[Beer81b] Beeri C. and Hoeyman P., ''Preserving Functional Dependencies,'' *SIAM J. on Computing*, 10:3,1981.

[Beer81c] Beeri C. and Vardi Y., ''On the Properties of Join Dependencies,'' *Advances in Data Base Theory*, 1, Gallaire H., Minker J., Nicolas J.M.,Plenum Press, 1981.

[Beer83] Beeri C., Fagin R., Maier D., and Yannakakis M., ''On the Desirability of Acyclic Database Schemes,'' *J. ACM*, 30:3, 1983.

[Beer84a] Beeri C., ''On the Membership Problem for Functional and Multivalued Dependencies in Relational Databases,'' *ACM Transactions on Database Systems*, 5:3,1980.

[Beer84b] Beeri C., Dowd M., Fagin R., and Statman R., ''On the Structure of Armstrong Relations for Functional Dependencies,'' *J. ACM*, 31:1, 1984.

[Beer86a] Beeri C., and Kifer M., ''An Integrated Approach to Logical Design of Relational Database Schemes,'' *ACM Transactions on Database Systems*, 11:2,1986.

[Beer86b] Beeri C., and Kifer M., ''Elimination of Intersection Anomalies from Database Schemes,'' *J. ACM*, 33:3, 1986.

[Bern76] Bernstein P.A., ''Synthesizing Third Normal Form Relations from Functional Dependencies,'' *ACM Transactions on Database Systems*, 1:4,1976.

[Bitt85] Bitton D., Mannila H., and Räihä K.-J., ''Design-By-Example: A Design Tool for Relational Databases,'' *Technical Report 85-692*, Cornell University, 1985.

[Bitt87] Bitton D., Millman J.C., and Torgersen S., ''A Logical Design Tool for Relational Databases,'' *Technical Report* , University of Illinois at Chicago, 1987.

[Brow85] Brown M.H. and Sedgewick R, ''Techniques for Algorithms Animation,'' *IEEE Software*, 1,1985.

[Carl76] Carlson C.R. and Kaplan R.S., ''A Generalized Access Path Model and Its Application to a Relational Database System,'' *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1976.

[Ceri86] Ceri S. and Gottlob G., ''Normalization of Relations and Prolog,'' *ACM Communications*, 29:6,1986

[Chan87] Chan E.P.F. and Hernandez H.J., ''On Designing Database Schemes Bounded or Constant-time-maintainable with respect to Functional Dependencies,''

[Chas81] Chase K., ''Join Graphs and Acyclic Data Base Schemes,'' *Proc. Int. Conf. on Very Large Data Bases*, 1981.

[Chen79] Chen P.P., ''The Entity-Relationship Model - Toward a Unified View of Data,'' *ACM Transactions on Database Systems*, 4:4,1979.

[Codd70] Codd E.F., ''A Relational Model of Data for Large Shared Data Banks,'' *Communications ACM*, 13:6, 1970.

[Codd72] Codd E.F., ''Further Normalization of the Data Base Relational Model,'' *Data Base Systems*,(R. Rustin, ed.), Prentice-Hall Englewood Cliff, NJ., 1972.

[Codd74] Codd E.F., ''Recent Investigations in Relation Data Base Systems,'' *Information Processing 74*,1974.

[Codd79] Codd E.F., ''Extending the Database Relation Model to Capture More Meaning,'' *ACM Transactions on Database Systems*, 4:4,1979.

[Codd85] Codd E.F., ''An Evaluation Scheme for Database Management Systems that are claimed to be Relational,'' , 1985.

[Come78] Comer D., ''The Difficulty of Optimum Index Selection,'' *ACM Transactions on Database Systems*, 3:4,1978.

[Cosm84] Cosmadakis S.S. and Kanellakis P.C., ''Functional and Inclucion Dependencies: A Graph Theoretic Approach,'' *Proc. of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1984

[Dega80] Degano P., Lomanto A., and Sirovich F., ''On Finding the Optimal Access Path to Resolve a Relational Data Base Query,'' *Proc. on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 88, Springer-Verlag, New York, 1980.

[Delo73] Delobel C. and Casey R.G., ''Decomposition of a Data Base and the Theory of Boolean Switching Functions,'' *IBM J. Research and Development*, 17:5, 1973.

[Delo74] Delobel C. and Leonard M., ''The Decomposition Process in a Relational Database,'' *Proc of Int. Workshop on Data Structure Models for Information Systems*, Belgium, 1974.

[Delo78] Delobel C., ''Normalization and Hierarchical Dependencies in the Relational Data Model,'' *ACM Transactions on Database Systems*, 3:3,1978.

[Derr85] Derrett N., Kent W., and Lyngbaek P., ''Some Aspects of Operations in an Object-Oriented Database,'' *IEEE Database Eng. Bull.*, 8:4,1985.

[Died88] Diederich J., and Milton J., ''New Methods and Fast Algorithms for Database Normalization,'' *ACM Transactions on Database Systems*, 13:3,1988.

[Dubo86] Dubois E., Hagelstein J., Lahou E., Rifaut A., and Williams F., ''A Data Model for Requirements Analysis,'' *Proc of Int. Conf. on Data Engineering*, 1986.

[Elma85] Elmasri R., Hevner A., and Weeldreyer J., ''The Category Concept: An Extension to the Entity-Relationship Model,'' *Data Knowledge Engineering 1.*, 1,1985.

[Fagi77a] Fagin R., ''Multivalued Dependencies and a New Normal Form for Relational Databases,'' *ACM Transactions on Database Systems*, 2:3,1977.

[Fagi77b] Fagin R., ''The Decomposition Versus the Synthetic Approach to Relational Data Base Design,'' *Proc. Int. Conf. on Very Large Data Bases*, 1977.

[Fagi79] Fagin R., ''Normal Forms and Relational Oprerators,'' *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1979.

[Fagi82a] Fagin R., ''Armstrong Databases,'' *IBM Research Report RJ3440*,San Jose, California, 1982.

[Fagi82b] Fagin R., ''Horn Clauses and Database Dependencies,'' *J. ACM*, 29:4, 1982.

[Fagi82c] Fagin R., Mendelzon A.O., and Ullman J.D., ''A Simplified Universal Assumption and Its Properties,'' *ACM Transactions on Database Systems*, 7:3,1982.

[Fagi82d] Fagin R. and Vardi M.Y., ''Armstrong Databases for Functional and Inclusion Dependencies,'' *Information Processing Letters 16*, 1983.

[Fisc83] Fischer P.C., Hou J.H., and Tsou D.M., ''Succinctness in Dependency Systems,'' *Theoretical Computer Science*, 24, 1983.

[Gali82] Galil Z., ''An Almost Linear-Time Algorithm for Computing a Dependency Basis in a Relational Database,'' *J. ACM*, 29:1, 1982.

[Gare79] Garey M.R. and Johnson D.S., *Computers and Intractability. A Guide to the Theory og NP-Completeness*, W.H. Freeman and Company, San Francisco, CA, 1979.

[Good83] Goodman N. and Shmueli O., ''Syntactic Characterization on Tree Database Schemas,'' *J. ACM*, 30:4, 1983.

[Gott87] Gottlob G., ''Computing Covers for Embedded Functional Dependencies,'' *Proc. Sixth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1987.

[Golu80] Golumbic M.C., *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[Grah79] Granham M.H., ''On the Universal Relation,'' *Computer Systems Research Group Report*, University of Toronto, 1979.

[Gran82] Grant J. and Jacobs B.E., ''On the Family of Generalized Dependency Constraints,'' *J. ACM*, 29:4, 1982.

[Hagi79] Hagihara K., Ito M., Taniguchi K., and Kassami T., ''Decision Problems for Multivalued Dependencies in Relational Databases,'' *SIAM J. on Computing*, 8:2,1979.

[Hull87] Hull R. and King R., ''Semantic Database Modeling: Survey. Applications, and Research Issues,'' *ACM Computing Surveys*, 19:3,1987.

[Jou80] Jou J.H., ''Theory of Functional Relational Relation Schemes in Relational Databases,'' *Vanderbilt University Tech. Report CS-80-10*, Nashvill Tennessee, 1980.

[Kupe84] Kuper G.M., Ullman J.D., and Vardi M.Y., ''On the Equivalence of Logical Databases,'' *Proc. of the 3rd ACM PODS*, 1984.

[Lien81] Lien Y.E., ''Hierarchical Schemata for Relational Databases,'' *ACM Transactions on Database Systems*, 6:1,1981.

[Lien82] Lien Y.E., ''On the Equivalence of Database Models ,'' *J. ACM*, 29:2,1982.

[Leuc88] Leuchner J., Miller L., and Slutzki G., ''A Polynomial Time Algorithm for Testing Implications of a Join Dependency and Embodied Dependencies,'' *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1988.

[Lewi77] Lewis E.A., Sekino L.C., and Ting P.D., "A Canonical Representation for the Relational Schema and Logical Data Independence," *IEEE Computer Software and Application Conf.*, 1977.

[Lin89] Lin D., "Automatic Logical Navigation Among Relations Using Steiner Trees," *Proc of Int. Conf. on Data Engineering*, 1989.

[Ling85] Ling T., "A Normal Form for Entity-Relationship Diagrams," *Proc. of the Fourth Int. Conf. on the Entity-Relationship Approach*,1985.

[Lucc78] Lucchesi C.L. and Osborn S.L., "Candidate Keys for Relations," *J. of Computer and System Sciences*, 17:2,1978

[Lyng87] Lyngbaek P. and Vianu V., "Mapping a Semantic Database Model to the Relational Model," *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1987.

[Maie79] Maier D., Mendelzon A.O., and Sagiv Y., "Testing Implication of Data Dependencies," *ACM Transactions on Database Systems*, 4:4,1979.

[Maie80] Maier D., "Minimum Covers in the Relational Database Model," *J. ACM*, 27:4, 1980.

[Maie83a] Maier D., *The Theory of Relational Databases*, Computer Science Press, Rockville, MD, 1983.

[Maie83b] Maier D. and Ullman J.D., "Maximal Objects and the Semantics of Universal Relation Databases," *ACM Transactions on Database Systems*, 8:1,1983.

[Maie84] Maier D., Ullman J.D., and Vardi M.Y., "On the Foundations of the Universal Relation," *ACM Transactions on Database Systems*, 9:2,1984.

[Mann85] Mannila H. and Räihä K.-J., "Design by Example: An Application of Armstrong Relations," *Technical Report 85-665*, Cornell University, 1985.

[Mann87] Mannila H. and Räihä K.-J., "Dependency Inference," *Proc. Thirteenth Int. Conf. on Very Large Data Bases*, 1987.

[Melk80] Melkanoff M.A. and Zaniolo C., "Decomposition of Relational and Synthesis of Entity-Relationship Diagrams," *Entity-relationship Approach to System Analysis and Design*,(P.P. Chen ed.),North-Holland Publishing Company, 1980.

[Mill88] Millman J.C., "Query Design in Design By Example," *Master of Science Thesis*, University of Illinois at Chicago, 1988.

[Motr86] Motro A., "Constructing Queries from Tokens," *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1986.

[Mylo83] Mylopoulos J. and Wong H., "Some Features of the TAXIS Data Model," *Proc. Sixth Int. Conf. on Very Large Data Bases*, 1983.

[Nava83] Navathe S. and Chen A., "A Methodology for Database Schema Mapping from Extended Entity Relationship Models into the Hierarchical Model," *Entity-Relationship Approach to Software Engineering*,(G.C Davis ed.),North-Holland Publishing Company, 1983.

[Pare77] Paredaens J., "About Functional Dependencies in a Database Structure and Their Coverings," *Philips MBLE Lab. Report 342*, Belgium, 1977.

[Peck88] Peckham J. and Maryanski F., "Semantic Data Models," *ACM Computing Surveys*, 20:3,1988.

[Riss77] Rissanen J., "Independent Components of Relations," *ACM Transactions on Database Systems*, 2:4,1977.

[Sagi80] Sagiv Y., "An Algorithm for Inferring Multivalued Dependencies with an Application to Propositional Logic," *J. ACM*, 27:2, 1980.

[Scio81] Sciore E., "Real-World MVD's," *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1981.

[Scio82] Sciore E., "Complete Axiomatization of Full Join Dependencies," *J. ACM*, 29:2, 1982.

[Ship81] Shipman D.W., "The Functional Data Model and the Data Language DAPLEX," *ACM Transactions on Database Systems*, 6:1,1981.

[Silv81] Silva A.M. and Melkanoff M.A., ''A Method for Helping Discover the Dependencies of a Relation,'' *Advances in Data Base Theory*, 1, Gallaire H., Minker J., Nicolas J.M.,Plenum Press, 1981.

[Tarj84] Tarjan R.E. and Yannakakis M., ''Simple Linear-Time Algorithms to Test Chordallity of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs,'' *SIAM J. on Computing*, 13:3,1984.

[Teor86] Teorey T.J., Dongqing Y., and Fry J., ''A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model,'' *ACM Computing Surveys*, 18:2,1986.

[Tsou82] Tsou D.M. and Fischer P.C., ''Decomposition of a Relation Scheme into Boyce-Codd Normal Form,'' *1982 ACM-SIGACT Conf.*, 1982.

[Ullm82] Ullman J.D., *Principles of Database Systems (Second Edition)*, Computer Science Press, Rockville, MD, 1982.

[Vard83] Vardi Y., ''Inferring Multivalued Dependencies From Functional and Join Dependencies,'' *Acta Informatica*, 19, 1983.

[Whan86] Whang K.Y., Ammann A., Bolmarcich T., et al, ''Office-By-Example, An Integrated Office System and Database Manager,'' *IBM Research Report*, 1986.

[Yao85] Yao S., *Principles of Database Design: Vol 11, Logical Organizations.* Prentice-Hall, Englewood Cliff, NJ, 1985.

[Zani76] Zaniolo C., ''Analysis and Design of Relational Schemata for Database Systems,'' *Report UCLA-EBG-7669*, University of California at Los Angeles, 1976.

[Zani81] Zaniolo C. and Melkanoff M.A., ''On the Design of Relational Database Schemata,'' *ACM Transactions on Database Systems*, 6:1,1981.

[Zani82] Zaniolo C. and Melkanoff M.A., ''A Formal Approach to the Definition and the Design of Conceptual Schemata for Database Systems,'' *ACM Transactions on Database Systems*, 7:1,1982.