Eötvös Loránd University

Faculty of Informatics

A General Partition Data Model and a Contribution to the Theory of Functional Dependencies

PhD Dissertation

Author: András Molnár

Supervisor: Prof. Dr. András Benczúr

Head of the Department of Information Systems

Doctoral School of Informatics

Director: Prof. Dr. János Demetrovics

Doctoral Program: Information Systems

Program Director: Prof. Dr. András Benczúr

2007.

Contents

Part I

Preliminaries

Chapter 1

Introduction

This thesis considers some issues of database design and data modeling. A database is a structured and efficient means of managing (storing, retrieving, manipulating, etc.) data, which is especially useful for large amount of data and complex domains. Database theory has been improved quickly in the last decades and covers a wide area of fields, including database design with data modeling, logical and physical design, semantic integrity constraints, query languages, query optimization and indexing, security and access rights, parallelism and transaction handling. Databases are used in practice in almost all application domains. A database is often part of a complex information system, responsible for management of enterprise tasks. Database design is a key issue in the effectiveness and the usability of such a system. Structure of a database must be flexible enough to allow the representation of extreme real-world cases, not to constrain the input of any possible information. At the same time, the structure must be rigid, in order to be consistent, well manageable and effective. In the ideal case, the structure should not allow entering of invalid data (that has no real-world meaning) and so, protect the database from inconsistencies. Good database design needs a deep analysis of the real-world application domain and a well-founded theory supporting the proper modeling of the domain.

The concept of *data model* has basically two meanings. First, it denotes a database paradigm, an approach and a logical language to formulate a database schema, capturing the real-world structure. The well-known traditional *relational model* is one example of it. Recent fields include object-oriented, object-relational and semi-structured data models. The thesis is based on the relational data model, which will be briefly introduced in Chapter 2.

Based on the data model behind, a database management system (DBMS) provides an abstraction layer for managing data. A data definition, manipulation and query language in terms of the data model provides a higher-level way to operate so that the user or application programmer does not need to consider issues such as physical storage and query algorithms. On the other hand, it has many advantages on the DBMS side. For instance, a higher-level declarative user query language (like SQL, Structured Query Language) allows the system to generate different query plans and choose the optimal one, depending on different conditions.

The second meaning of term *data model* refers to a conceptual model for a particular application domain. It is necessary to identify real-world concepts during the design of a database system. Usually, entity classes are modeled with their properties and relationships. Different types of relationships among entity classes exist, depending on the number of the participating entity classes, on compulsory or optional participation of an entity in a relationship and on the cardinality of the related entities. Cardinality constraints and functional dependencies must be acquired in order to obtain the precise specification of a relationship. In many cases the schema can be decomposed in order to avoid redundancy and anomalies. This process is called schema refinement and normalization. The traditional syntax and axiomatization of constraints is, however, not 'user-friendly'. One of the contributions of this thesis is a representation and reasoning framework for one of the traditional constraint types, *functional dependencies*, allowing more effective specification of schemata and relationship types described by sets of constraints than the traditional approach. All the different possible sets of constraints for small arities (up to five) are computed. These patterns can be surveyed more deeply in the future and the approach may be extended with other types of constraints.

Data warehouses are used for handling large amount of data for archiving and analysis purposes. The core data model behind is the *multidimensional model*. Multidimensional databases have been playing a significant role in the database field. A main advantage of them is to provide the user a clear data structure and navigation operations that allow to view data in multiple levels of aggregation. It is usually assumed that all data is available in the finest granularity (*raw data*) and has more or less homogenous multidimensional structure. However, in some cases, the available data can have a heterogenous structure that comes either from the nature of the application or the incompleteness of known information. It is important in such cases to design a proper schema that exactly describes and relates the available datasets (extensional data), supports determining what information can be derived out of them (intensional data), what are the valid, meaningful user navigation and aggregation operations, and ensures the consistent extension of the database with further datasets as more information becomes available.

One of the key issues is *summarizability*, i.e. whether an aggregation value of a set can be obtained from pre-aggregates of its subsets, without the need of raw data. Disjointness of these subsets must be assumed but summarizability depends on other aspects as well, such as the type of aggregation.

This thesis proposes a formal basis for a data model that is based on the partition semantics of relations. It uses the well-known relational theory to describe structures of sets and their aggregation data, extends the notions of semantical constraints that allows specifying how base sets of relations relate to each other and determines what algebra operations are valid for instances. The first version of a basic partition language is sketched, allowing the management of set names and structures, based on disjointness. The abstract model can be concretized by adding particular 'plug-ins' into the langugage, depending on the application and the algebraic structure of the set names.

1.1 Overwiew of the Thesis

The thesis is separated into five parts. Part I continues with the introduction of database design issues, including the concept of *relationship* and *functional constraint* and shows the traditional formalism of functional constraints (Chapter 2). Chapter 3 briefly demonstrates the *multidimensional data model* and the concept of *geographical field* as examples of *partitioning*. It introduces some cases that need a more sophisticated data model than the traditional multidimensional model. Both chapters give a brief overview of the related contributions.

The main contributions are presented in Part II and III.

In Part II, Chapter 4 introduces a simple partition model, based on the *partition semantics of relations* with cumulative attributes. It discusses the role of functional dependencies for partition relations and proposes other type of constraints (*base set constraints*). Some problems with the naive usage of relational algebra operations are presented and a sound algebra is given for partition relations. Chapter 5 presents in more detail how the structure of partitions can be described using functional constraints and gives a graphical represenation, the *structural graph*. The notion is extended for the case of partial information and heterogenous base sets of partitions. Chapter 6 puts the partition model into a four-level database framework and establishes the *basic partition language* for schema construction and manipulation as well as querying. It gives powerful description management and aggregation administration facilities for disjoint set structures. Finally, some of the many open issues and improvement areas are sketched. The partition model may be improved further to support data warehouse construction or metadata handling of large amount of data assets.

Functional dependencies can be used to describe the structure of partitions (refinement relations). In general, they are widely used in database design as mentioned before. Part III is devoted to functional dependencies and negated functional dependencies. Chapter 7 proposes a simplified formalism and a powerful axiomatization by eliminating most of the redundancies of the traditional syntax and proves its soundness and completeness. Chapter 8 introduces a spreadsheet representation of constraint sets and discusses the number of different cases for small arities. Chapter 9 puts the representation into a graphical form and shows how the implication rules can be translated into graphical reasoning patterns. Further methods of managing constraint sets are considered in Chapter 10, including a survey on decompositions and the conversion into the partition structural graph notation as well.

Summary and conclusions are presented in Part IV. Appendices form the fifth part of the thesis.

1.2 Acknowledgments

The author is grateful to many people who helped and supported his work, including supervisor Prof. András Benczúr from Eötvös University Budapest, Prof. János Demetrovics and his colleagues from SZTAKI[1], Prof. Bernhard Thalheim from Christian-Albrechts-Universität Kiel, Prof. Hans-Joachim Lenz from Freie Universität Berlin, Attila Kiss and colleagues from the e-Science Regional Knowledge Center at Eötvös University, anonymous reviewers of papers, as well as the author's parents, sister Andrea Molnár, friends and other colleagues including Csaba Sidló. This thesis could not have been completed without their contribution. Many thanks for supervising, cooperation, valuable comments, motivation, patience and all other means of support.

[1]Computer and Automation Research Institute of the Hungarian Academy of Sciences.

Chapter 2

Relations, Relationships and Functional Constraints

Database design is the process of creating a proper database schema by analyzing and modeling real-world entities and processes, and constructing their representation and implementation in a faithful and efficient way. Specification of database structuring is based on three interleaved and dependent parts [64]:

Syntax: Inductive specification of structures uses a set of base types, a collection of constructors and a theory of construction limiting the application of constructors by rules or by formulas.

Semantics: Specification of admissible databases on the basis of static integrity constraints describes those database states which are considered to be legal.

Pragmatics: Description of context and intension is based either on explicit reference to the enterprise model, tasks, policy and environments or on intensional logics used for relating the interpretation and meaning to users depending on time, location, and common sense.

Specification of syntax is based on the database modeling language. In most cases a visual language is used to define the *conceptual model* of the database by modeling real world objects with their properties and relationships. This is then transformed into the *logical model*, formalized in terms of a representational model of the underlying database paradigm (e.g. relational tables or object classes). This can be mapped into the syntax of the data definition language of the chosen database management system, considering access rights, physical storage and other issues.

Specification of semantics requires a logical language for specification of classes of constraints. Typical constraints are dependencies such as functional, multivalued, and inclusion dependencies, or domain constraints [1]. Most of them are traditionally formalized in the relational data model. However, constraint specification must be based on modeling the real world on the conceptual level. Graphical languages have constraint specification facilities but they are usually limited and do not consider logical implications or consistency check in the schema. Moreover, specification of semantics is often rather difficult due to the complexity. For these reasons, it must be supported by a number of solutions supporting acquisition and reasoning on constraints.

Specification of pragmatics is often not explicit.

2.1 Relational Data Model

The relational data model [24] has been the most widely used database paradigm, due to its simplicity, expressive power, solid theoretical foundations and effective implementation possibilities [1, 42].

In relational database theory, the syntactical part of a *database schema* consists of finitely many *relational schemata*, and each relation schema has one or more (finitely many) *attributes*. It describes the structure of the data. Actual data, contained in a *database instance* consists of *relational tables* for each relation schema such that the attributes correspond to columns of the table. More precisely it means the following [1]

Let $Attr$ and Dom be countable sets of symbols and a function $DOM : Attr \to 2^{Dom}$. The universe of attribute names is $Attr$ and the universe of database (content) symbols is Dom. DOM assigns a type (domain) to each attribute. The concept of *relational schema* is defined as a finite set of functions $r : \mathcal{R} \to Dom$. R is by definition a *relational instance (relation)* of schema $\mathcal{R}$ if and only if $\forall r \in R : \forall A \in \mathcal{R} : r(A) \in DOM(A)$. Columns (attributes) of R are the elements of $\mathcal{R}$, while rows of R are the elements of R itself.

Let Rel be a countable set of symbols (*names of relational schemata*) and T its finite subset. A function $\mathcal{DB} : T \to 2^{Attr}$ is a *relational database schema*. A function DB is a *relational database (instance)* of schema $\mathcal{DB}$ if it maps each element t of T to a relation of schema $\mathcal{DB}(t)$. These relations are the tables of the database, set T contains the names of tables.

Example 1. *For instance, let relation J contain a daily flight schedule. Its relational schema can be $\mathcal{J} = \{Flight, From, Destination, Time\}$ and one of its tuples may be $r = \{Flight \mapsto 6209, From \mapsto Budapest, Destination \mapsto London, Time \mapsto 18:05\}$. Fixing the order of attributes simplifies the notation: $r = (6209, Budapest, London, 18:05)$.*

If pilots of flights are incorporated into the database, we get a database schema with three tables (relation schemata): $T = \{\mathcal{P}, \mathcal{J}, \mathcal{V}\}$ where $\mathcal{P}$ is the schema of pilots and $\mathcal{V}$ stands for a schema whose instance will store the assignment of pilots to flights.

A relational schema can be extended with semantical *integrity constraints* to ensure consistency of a database by specifying which instances are considered as valid databases. The database management system checks that the prescribed constraints are not violated by any transaction.

Functional dependencies are probably the most known database constraints. For two sets of attributes X, Y of a relational schema R, $X \to Y$ states that only those tables are treated as valid instances of R that contain no pair of rows that have the same values in the columns of attributes X but differ in any of the columns of attributes Y, i.e. the values of X uniquely determine the values of Y. For instance, in an address table of schema $(City, ZIP, Street, HouseNr)$, the ZIP code determines $City$ so there is a functional dependency $ZIP \to City$ between them. See more on functional dependencies in Section 2.3.

2.2 Conceptual Modeling: Entities and Relationships

The *Entity-Relationship (ER) model* (e.g. [22, 23, 21, 73]) is a popular graphical tool for conceptual data modeling. The design procedure is based on identification of entity classes and relationships among them. Entity classes are represented as rectangles and relationship classes as diamonds connected to the participating entity classes. Relationships are usually binary, i.e. two entity classes participate in the relationship. Relationships of higher arity (e.g. ternary, quaternary) have more participating entity classes. They are also allowed and should be used whenever convenient.

[1] There are other equivalent definitions [1]. Here, attributes are treated as typed, identified by their names. Therefore, their order in the schema is arbitrary and the definition is given in a functional fashion.

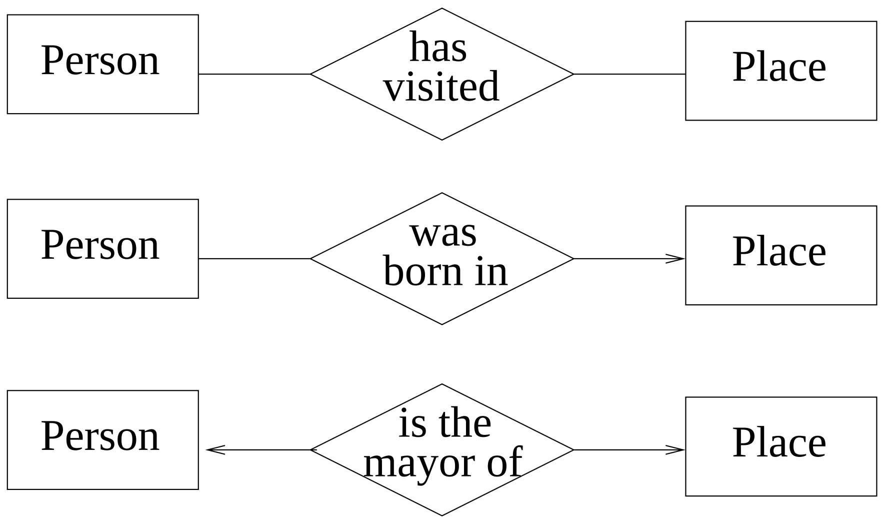


Figure 2.1: Three basic types of binary relationships in the Entity-Relationship model (arrowheads represent cardinality restrictions similarly to functional dependencies)

The model allows specification of cardinalities of entities participating in relationships. The focus here is on relationships that can be described by sets of functional dependencies. Therefore, the only possible cardinalities considered are *'at most one'* (usually denoted by $(0,1)$) and *'any'* $(0,n)$. They refer to how many times an entity can participate in a particular relationship (whether it can relate to more than one entity of the other class or not). A simple notation is an arrowhead on the opposite side to represent a $(0,1)$ cardinality. It harmonizes with the functional dependency notation: an entity that participates only once uniquely determines the entity it is related to via the relationship.

The three basic types of binary relationships that can also be expressed by functional constraints are *many-to-many, many-to-one and one-to-one*. Figure 2.1 shows an example for each type. In the first relationship, any person can visit any number of places (villages/towns/cities) and a place can be visited by an unrestricted number of people. It is a many-to-many relationship that can be specified by the empty set of functional dependencies. In the second case, a person was born at exactly one place but several persons may have been born at the same place and so it is a many-to-one relationship corresponding to the singleton functional dependency set $\{Person \rightarrow Place\}$. In the third case, assume a place has currently one mayor and a person can be the mayor of only one place so there is a one-to-one relationship between them: $\{Person \rightarrow Place, Place \rightarrow Person\}$. Furthermore, if we allowed a person to be the mayor of more than one place at the same time, we would get a *one-to-many* relationship as a fourth case. However, it is basically of the same type as many-to-one by changing the roles of the two participating entity classes.

A relational database schema from an ER graph can automatically be generated [42]: entity and relationship classes are transformed to relational schemata with attributes and integrity constraints. If relationship types are exactly specified in the above terms, this transformation process can optimize the generated schema by eliminating redundancies (schema normalization).

The general entity-relationship model allows more sophisticated specification of cardinalities like optionality and exact numbers instead of the notation n. In the above examples, one may specify that a place must have a mayor. However, *inclusion dependencies* would be needed in addition to functional dependen-

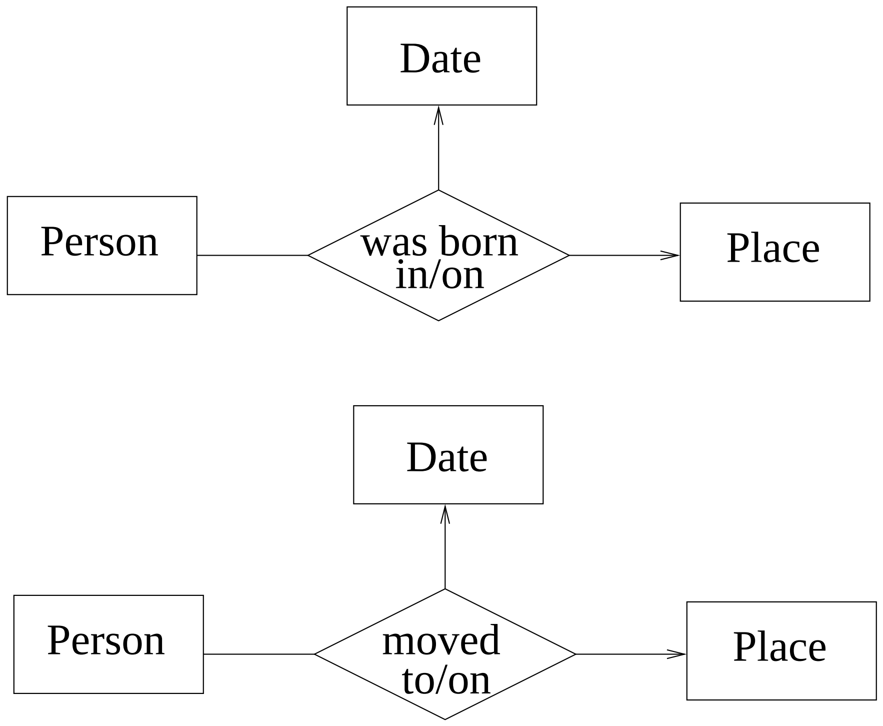


Figure 2.2: Ambiguity of ternary relationship specification in the Entity-Relationship model

cies to describe that an entity must participate in a relationship (e.g. for a $(1, n)$ or $(1, 1)$ cardinality).[2] Therefore, optionality is not considered by us as an aspect of relationship type. Furthermore, the general entity-relationship language allows specification of cardinalities like $(2, 3)$ (at least 2, at most 3) that are not captured by functional constraints. However, the variety of relationship types is already rich for higher arities if we consider only the simple case of relationships described by sets of functional dependencies (see Chapter 8).

Consider the two examples on Figure 2.2. In the first case, each person can have one birth date and place. In the second case, each person can have moved several times but only once on a day and only once to a specific place (the example assumes nobody can move back to a previous place – or if so, previous dates of moves to the same place are not stored). It is clear that the structure of the two ternary relationships essentially differs, and this difference is not captured by the entity-relationship notation and other common graphical tools.[3] The first case has the functional dependency set $\{Person \rightarrow Date, Person \rightarrow Place\}$ while the second has $\{PersonPlace \rightarrow Date, PersonDate \rightarrow Place\}$. In a later phase of schema design, they can be decomposed to binary relationships. However, *binarization* of relationships usually needs introduction of extra relationship or even entity classes (e.g. as in the second case of our example). While binarization can

[2]Anyway, if the relationship is transformed into a relational schema, inclusion depencencies are required as referential integrity. However, this is a conceptually different phenomenon.

[3]Different versions of the original cardinality constraint notation can be used (instead of the arrows) as participation or *look-through* constraints [73]. However, neither of them is capable to describe precisely all the possible ternary relationship structures.

be useful in some cases, it makes the graph notation more complex and can make the schema less natural and perspicuous already in a relatively early stage of the database design process.

Since designing a relationship of higher arity can be incomplete or ambiguous, binarization is often performed even if a relationship of higher arity would provide a more suitable model. In fact, complexity of these kind of relationships can be high and the different types of ternary, quaternary relationships are not considered (as opposed to the well-known three types of binary cases). The complete and unambiguous specification can be done by recalling database constraints, and relationship construction can be achieved through relation schema design. To achieve this, the database developer must master semantics acquisition.

2.3 Functional Dependencies and Excluded Functional Constraints

The notion of functional dependencies was introduced in [5] for the relational database model to provide a way for specification of the properties of valid, acceptable instances of a relational schema. Since then, the theory of functional dependencies has been well developed. Classical database design is based on a step-wise extension of the constraint set and on a consideration of constraint sets through generation by tools. Dependency theory at schema design is usually applied for determining keys and decomposing schemata into normal forms (e.g. [9, 10, 8, 11]).

2.3.1 Traditional Concepts and Notations

Let $\mathcal{R}$ be a relational schema. If X and Y are two subsets of its attributes then $X \rightarrow Y$ is a *functional dependency* for $\mathcal{R}$ and is said that X functionally determines Y.

A relation R of schema $\mathcal{R}$ *satisfies* the functional dependency (denoted by $R \models X \rightarrow Y$) by definition if $\forall r, s \in R : (\forall A \in X : r(A) = s(A)) \Rightarrow (\forall B \in Y : r(B) = s(B))$ (or simply $r(X) = s(X) \Rightarrow r(Y) = s(Y)$). Satisfaction can be naturally generalized for sets of constraints.

Notation XY is used instead of $X \cup Y$ for two sets of attributes X and Y. For a single attribute A, the set $\{A\}$ will be denoted by A when it causes no confusion.

It is an important task during schema design to determine which of the possible functional dependencies are valid, in order to avoid update anomalies and find a suitable decomposition of the relational schemata.

A *key* is a set of attributes that functionally determines all attributes of the relation schema. There may be several keys w.r.t. a set of functional dependencies.

Invalidity of some possible functional dependencies may also be recognized during schema design. This information should be declared together with valid functional dependencies. Besides functional dependencies (FDs), *excluded functional constraints* (also called *negated functional dependencies*) are considered as well: e.g. $X \not\rightarrow Y$ states that the functional dependency $X \longrightarrow Y$ is not valid.[4]

Specified constraints are not independent of each other: one or more constraints may logically *imply* other constraints while some possible constraints remain independent. If a constraint $X \rightarrow Y$ holds in each possible schema where a set of constraints $\mathcal{F}$ holds, it is denoted by $\mathcal{F} \models X \rightarrow Y$. More precisely, for each relation R satisfies $\mathcal{F}$, R satisfies $X \rightarrow Y$ as well.

For a finite set $\mathcal{F}$ of functional constraints (both positive and negated), its *closure* ($\mathcal{F}^+$) is defined as the constraint set containing all implied constraints of $\mathcal{F}$. In most cases, the focus is on *closed* sets of functional dependencies. $\mathcal{F}$ is closed if and only if $\mathcal{F}^+ = \mathcal{F}$. Due to trivial constraints, the closure implicitly depends on the underlying relational schema $\mathcal{R}$. This definition will be simplified in Chapter 7.

[4]Negated functional dependencies are interpreted by the *weak semantics*, i.e. a negated dependency is allowed to be valid in certain instances but it is not the general case for instances of the schema. An alternative, *strict semantics* leads to the definition of *afunctional dependencies* [12]. In fact, implication of constraints is equivalent for both semantics [35].

An *axiomatization* of functional constraints consists of axioms and rules that support reasoning on constraints. If an implication can be derived formally, using an axiomatization $\mathcal{A}$, it is denoted by $\mathcal{F} \vdash_{\mathcal{A}} X \to Y$. $\mathcal{A}$ is *sound* if each derivation is correct ($\models$ follows from $\vdash_{\mathcal{A}}$) and is *complete* if each logical implication can be derived using $\mathcal{A}$ (i.e. $\models$ implies $\vdash_{\mathcal{A}}$). Extension of the notation for negated constraints is straightforward.

2.3.2 Constraint Set Development

A main task is to determine the validity of all possible functional dependencies given an initial set, i.e. to get the closure of a constraint set. This constraint acquisition process is usually performed by a step-wise extension of the constraint set, both for positive and negated constraints:

The set of valid functional dependencies Σ_1***:*** All dependencies that are known to be valid and all those that can be implied from the set of valid and excluded functional dependencies.

The set of excluded functional dependencies Σ_0***:*** All dependencies that are known to be invalid and all those that are invalid and can be implied from the set of valid and excluded functional dependencies.

This pragmatical approach leads to the following simple elicitation algorithm illustrated by Figure 2.3.:

1. *Basic step: Design obvious constraints.*
2. *Recursion step: Repeat until the constraint sets* Σ_0 *and* Σ_1 *do not change:*
   - *Find a functional dependency* α *that is neither in* Σ_1 *nor in* Σ_0.
     If α *is valid then add* α *to* Σ_1. *If* α *is invalid then add* α *to* Σ_0.
   - *Generate the logical closures of* Σ_0 *and* Σ_1.

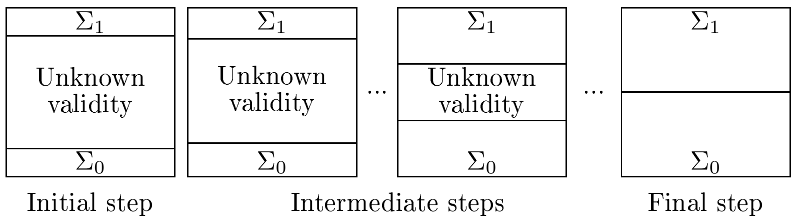


Figure 2.3: Constraint Acquisition Process

2.3.3 Traditional Axiomatization

The Extended Armstrong Implication System

Functional dependencies and excluded functional constraints are axiomatizable by the following system [73].

Axioms

$$XY \to Y$$

Rules

$$(1)\ \frac{X \longrightarrow Y}{XVW \longrightarrow YV} \qquad (2)\ \frac{X \longrightarrow Y\,,\quad Y \longrightarrow Z}{X \longrightarrow Z}$$

$$(3)\ \frac{X \longrightarrow Y\,,\quad X \not\longrightarrow Z}{Y \not\longrightarrow Z}$$

$$(4)\ \frac{X \not\rightarrow Y}{X \not\rightarrow YZ} \qquad\qquad (5)\ \frac{XZ \not\rightarrow YZ}{XZ \not\rightarrow Y}$$

$$(6)\ \frac{X \longrightarrow Z\,,\quad X \not\rightarrow YZ}{X \not\rightarrow Y} \qquad\qquad (7)\ \frac{Y \longrightarrow Z\,,\quad X \not\rightarrow Z}{X \not\rightarrow Y}$$

Let us consider an example of a derivation with the system just presented:

Example 2. *Assume a relational schema $\{Flight, Pilot, Time\}$ describes the assignment of flights with pilots for certain times. The schema prescribes that a pilot at a certain time can drive only one flight: $Time\,Pilot \rightarrow Flight$. We also assume a flight can be driven by only one (chief-)pilot at a certain time: $Flight, Time \rightarrow Pilot$. Furthermore, if a flight is assumed to fly only once, the constraint $Flight \rightarrow Time$ must be added to the set. Consider implications with the rules above. As a first step, $Flight\,Pilot \rightarrow Time$ is derived using the extension rule (1). Note that no non-trivial implication can be derived using the axiomatization. For instance, the extension rule yields $Flight \rightarrow Flight\,Time$. Although this is an irrelevant constraint, it must be derived in order to use the transitivity rule (2). Its result is $Flight \rightarrow Pilot$.*

Is the set closed now in the example? Of course not, since the following constraints can be derived: $Flight \rightarrow Flight$, $Flight \rightarrow Flight\,Pilot$, $Flight \rightarrow Time\,Pilot$, ... Although these are trivial and redundant constraints (they do not hold new information), one of them may be needed for a non-trivial derivation. It is not easy to see whether a non-trivial derivation exists. Such trivial and redundant constraints make constraint acquisition unnecessarily complicated and the aim of a new implication system will be to get rid of them.

2.3.4 Representation by Closed Attribute Sets

An alternative approach of representing constraint sets is based on closed attribute sets.

Given a set $\mathcal{F}$ of functional dependencies, the *closure of an attribute set* X, denoted by $X^{+(\mathcal{F})}$ is the maximal set of attributes Y for which $X \rightarrow Y$ holds (implied by $\mathcal{F}$). An attribute set X is *closed* if and only if $X = X^{+(\mathcal{F})}$.

The set of closed attribute sets w.r.t. a set of functional dependencies is closed under intersection so they form an *intersection-semilattice* (meet-semilattice, SL, [30, 29, 28]). This can be reversed: each set of attribute sets containing the full set (of all the attributes of the schema) that is closed under intersection forms an SL and there exists a set of functional dependencies whose closed attribute sets are exactly the items of the SL.

The semilattice can be represented by a graph [26, 18]: labeled nodes correspond to closed attribute sets and the (nontransitive instances of) set containment are represented by edges. Since the attributes are 'inherited' along the edges, it is enough to indicate the new attributes at each node only. For the sake of clarity, all the attributes will be indicated here with the inherited ones put in brackets. Examples of semilattice graphs can be found in Chapter 5.

The upper bound of the semilattice is the full set of attributes while the lower bound is usually the empty set (the lower bound contains the constant attributes if there are any). Neighbours of the full set are called *coatoms* or *antikeys* ([75], [37], [26]). Equivalent attributes (A, C such that $A \rightarrow C$ and $C \rightarrow A$ both holds) always occur together in a node.

The semilattice graph corresponds to a different point of view than the traditional notation: the more positive dependencies we have, the less is the number of closed attribute sets. Although we may have redundant dependencies, adding a nonredundant dependency always removes at least one closed set. The semilattice graph notation may become complex if the set of dependencies is simple. Therefore, by designing or surveying a relation schema, both representations can be used in parallel, focusing on the one more convenient.

2.3.5 Decomposition and normal forms

As soon as the complete specification of the set of functional dependencies is obtained (i.e. the validity of each possible dependency is marked as either positive or negative), their impact must be considered in the schema. This leads to the decomposition of the schema in many cases, to avoid redundancy. The aim is usually to reach a normal form by lossless decomposition. Considering only functional dependencies it can be either BCNF or 3NF [1, 42] (see Chapter 10).

There are well-known normalization algorithms [42]. It is important to note that normalization is based on the completeness of constraint sets. Another issue is the nondeterminism of normalization algorithms. The result may depend on the order of the constraints. A better approach for small arities may be to survey the possible normalized decompositions and let the designer choose among them, based on other practical or semantical considerations. Such a survey for ternary cases can be found in Section 10.3. A similar survey will be interesting for quaternary cases as well.

2.4 Contribution to the Theory of Functional Constraints

More details on the contributions can be found in [31, 32, 34, 36].

2.4.1 Representation and Reasoning

As seen above, the traditional formal notation considers dependencies one-by-one, including trivial and redundant ones. The implication is not effective enough in most cases. There is usually a strong interdependence among constraints that is not visualized. All these lead to an inconvenience in using the formalism with the traditional axiomatization. Although the well-known closure algorithm [1] can be effectively used to derive the logical implications for a specific set of attributes, it cannot deal with negated constraints and do not highlight the logical connections or the impact of adding or deleting a constraint from a set. Therefore, simple and sophisticated means of representation and reasoning for constraint sets are needed. Solution of the problem is crucial since typical algorithms such as normalization algorithms can only generate a correct result if specification is complete, i.e. each possible constraint is declared as either valid or invalid in a consistent, non-contradictory way.

Chapter 7 will show how the traditional syntax can be simplified by invalidating trivial and redundant constraints. A simple and powerful axiomatization called ST is given for functional dependencies. It is extended with additional rules for negated constraints (PQRST ruleset). Soundness and completeness over the universe of singleton and nontrivial constraints are proven by simulation. There exists an order of rule application that yields a complete algorithmic method for deriving all implied constraints. This simplified syntax and axiomatization is the formal basis for spreadsheet and graphical representation and reasoning presented in Chapters 8 and 9.

There have been several proposals for graphical representation of sets of functional dependencies (e.g. [6, 77, 16]). Nevertheless, these graphical notations have not made there way into practice and education. The main reason for this failure is the complexity of representation. Chapter 9 proposes an approach for graphical reasoning of sets of functional constraints for small relation schemata by nodes of geometrical figures and graphical rule patterns.

An alternative, spreadsheet-based representation is presented in Chapter 8. The proposal of both representations of constraint sets provides a possible solution of the problem of defining a pragmatical approach that allows simple representation and reasoning on database constraints.

Section 10 considers some miscellaneous topics and methods regarding to reasoning on constraint sets.

2.4.2 Number of Constraint Sets

In fact, the number of constraints may be exponential in the number of attributes [27]. Specification of the complete set of functional dependencies may be a task that is infeasible. This problem is closely related to another well-known combinatorial problem presented during MFDBS'87 [72] and that is still only partially solved: What is the size of sets of independent functional dependencies for an n-ary relation schema?

Although the number of different relationship types for n attributes is still unsolved[5], one of the main contributions of this thesis is to generate and compute the different constraint sets and cases for 3, 4 and 5 attributes. Section 8.2.4 shows the high complexity for these small arities. It is not surprising from the point of view of constraint sets. However, a relationship type can be described by a constraint set and most works that consider non-binary relationships give surprisingly small attention to this complexity. In fact, some notations used in practice are ambiguous.

[5]Estimations exist, see [14, 30]

Chapter 3

Dimensions and Partitions

Partitioning of sets is a widely used concept in database theory and in applications. A partitioning is usually supposed to be given in terms of the partitioned elements (the element data is assumed to be given) and discussed in an application-oriented framework, e.g. geographical information systems (GIS, [68]) or on-line analytical processing (OLAP, [47, 59, 20, 49]). The general concept of partitioning and modeling the structure of multiple partitions with different levels of information is rarely considered.

This chapter presents the concept of geographical fields and the multidimensional data model briefly and informally, followed by the basics of aggregation. Based on all these, it gives an overview of motivating cases that point towards a general partition data model.

3.1 Geographical Fields

Many aspects of GIS data is about partitioning a part of the geographic space. Just a brief introduction is given of how a geographic space can be interpreted as a partition. For more details, see papers such as [39, 40, 67], which successfully apply the partition concept in the theory of spatial data management.

Modeling the geographical space can be either entity-based or field-based [68]. In an entity-based model, several spatial object classes are identified (e.g. cities, countries, rivers, streets, buildings, etc.) and populated with spatial objects having their geometry and attributes.

In a field-based model, one or more of the variables (e.g. altitude, temperature, land use) are identified. They are functions over the points of the geographic space. Their value types can be classificational (discrete) or continuous. A discrete field can be viewed as a partition of the space where two points belong to the same subset if and only if their field values are equal.

Figure 3.1 shows an example of the conceptual model of a geographical field, drawn in the ArgoCASEGEO graphical language [41], an extended UML syntax with spatial stereotypes as pictograms. *Land use* is modeled as a geographical field, a function over points of the geographical space taking its values of land use types (e.g. road, building, forest, water, etc). The triangle with letter F refers to geographic field stereotype and the field type pictogram indicates that the field is represented as a polygon partition. Land use type objects are non-geographical objects (indicated by an empty triangle) referenced by the polygons.

A spatial database is usually made up from several datasets[1] corresponding to classes of spatial entities or geographical fields forming thematic layers. An important operation is the *overlay* or *spatial join* when

[1]A dataset may be a relational table or objects of a class, based on the representational data model used.

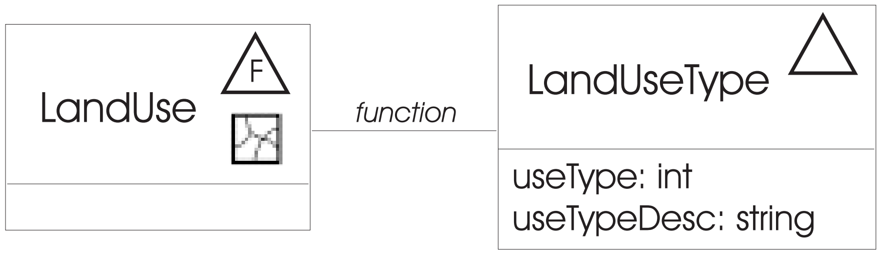


Figure 3.1: Land use modeled as a geographical field

different thematic layers are joined based on geographical location. An overlay of partitions constructs a new partition by generating each possible intersections between the two partition [68]. The information often comes from different sources and the overlay becomes error-prone. Errors can be discovered from a semantical point of view: topological rules may define constraining relationships among field values (e.g. the border of a country shall not divide a city polygon).

The concept of partitioning often appears in entity-based models as well. The well-known *topological data model* (e.g. TIGER/LINES, [68]) is based on a polygon topology made up from elementary geometries resulting from the overlay of all entity classes. A planar graph is constructed from nodes and lines of original geometries (points, polylines and polygon boundaries), extended with nodes created from each possible intersection. Edges and nodes of this graph together with the elementary planar regions they define form the polygon topology. Geometry of each geographical entity is referenced in terms of the polygon topology (e.g. a polygon can be made up of several neighboring elementary regions). If two polygons have identical boundary sections (e.g. a county and a country) they share their common geometry in the polygon topology. The shared geometries may allow to determine the intersecting items of different thematic layers without the need of geometrical operations.

A polygon topology or a field overlay can be viewed as a multiway partition of the space.

The following section introduces the multidimensional data model that can be applied for geographical fields as well.

3.2 Data Cubes in the Classical Multidimensional Data Model

Multidimensional databases have been playing a significant role for the last years in the database field, especially in analytical processing (OLAP) and data warehousing (e.g. [20, 45, 47]). Items of a population (e.g. transactions, inventory items, phone calls) are characterized by dimensions (e.g. time, type, amount, location) in different granularities (e.g. day, week, month, year for time).

A *data cube* stores aggregated (summarized) values of items in its cells, corresponding to different values of its dimensions in any chosen granularity. A *cuboid* is a cube view for a particular combination of dimension granularities. The finest known granularity is the base data (raw data), or *base cuboid*. Other cuboids can be generated based on this and form a lattice [7]. To enhance query processing speed, some cuboids may be stored as materialized views.

The schema of a cuboid can be viewed as a multiway partitioning over the base set of items, cells correspond to the subsets generated by the overlay of partitions according to the dimension granularities of the cuboid, storing summary data of each subset.

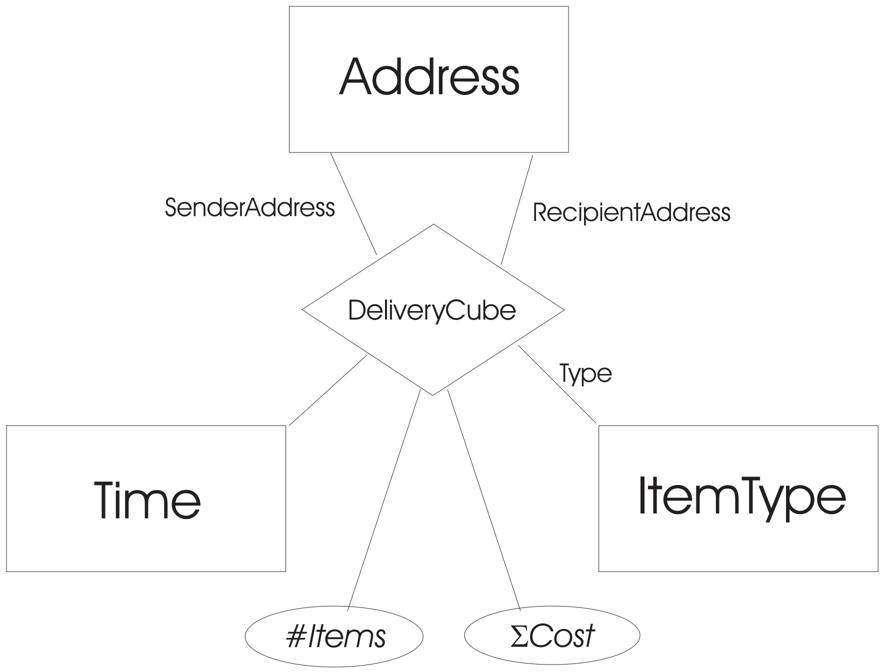


Figure 3.2: A star schema: Model of the data cube of Example 3, without details of dimensions

A relational approach of multidimensionality is the *star schema*, where a *fact table* contains the items as rows with their non-dimensional attributes and foreign keys referring to dimension tables with dimensional attributes [47]. The fact table can be aggregated according to selected dimensional attributes to obtain aggregated data in different granularities. Based on the structure of dimensions, a normalization process may lead to a *snowflake schema* where some dimension tables are decomposed – however, denormalization is considered in many cases due to efficiency and ease of use but there is a debate on that [63].

Two examples are presented for later reference:

Example 3. *Assume a postage agency has a database of delivered items. Each item is classified by the following aspects, forming the dimensions of the schema. The possible values are indicated in brackets for some attributes. Attributes form dimension hierarchies in the order of their appearance, i.e. they correspond to different granularities.*

Type: *T(telegraph/postcard/letter/package)*

Time: *Year(2006/2005/...), Month(Jan/Feb/...), Day(1/2/...)*

SenderAddress and RecipientAddress: *Country, State, Region, County, City, District, ZIP, Street, HouseNr*

The aggregated values represent the number of posted items and their total cost. The conceptual model of the cube is drawn of Figure 3.2 without the details of dimensions.

A three-dimensional cuboid of summarized values can be constructed for example for types, months and recipient cities, aggregated over the sender address. It is sketched on Figure 3.3.

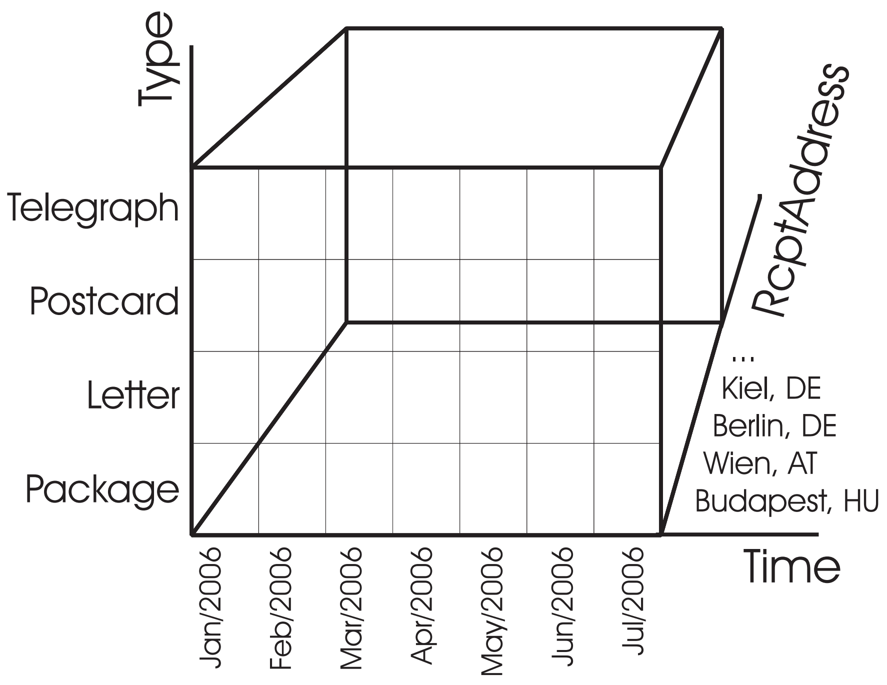


Figure 3.3: A classical data cuboid of Example 3

Example 4. *Assume a public transport company keeps track of the journeys with the help of a chip-card ticketing system. The following dimensions are available, based on attributes of journeys:*

TimeStart,TimeEnd: *Year, Month, Day, Hour, Minute*

StationStart,StationEnd: *Code, Name, Zone*

LineNr *,*

Vehicle: *Id, Type(bus/train/boat)*

Ticket: *SerialNo, Type(single/travelcard)*

The aggregated values represent the number of journeys for each valid value combination.

A data cube is an effective, easy-to-use model for data analysis. In a classical case, the complete data is available in the finest possible granularity (by item) and dimensions have a clear hierarchical structure. This hierarchy provides navigation paths for viewing the data. Starting from the example cuboid described in Example 3, the user can navigate along each dimension to see data in different levels of detail. Starting from the aggregated view by month, the following operations are valid for the *time* dimension as an example:

Roll up: Rolling up on *time* aggregates over *months* and the result is a cuboid by *years*, a coarser partitioning over items. The values are summarized for each year.

Drill down: Drilling down on *time* unfolds the cuboid of *months* and refines the partitioning to *days*. The summarized values are replaced by detailed values.

Slice: Slicing a cuboid means selecting a part of it. A possible slicing for dimension *time* on a monthly level is selecting cells that correspond to the months Jan from Jul 2006 as seen on Figure 3.3.

Another usual operation is *dice*, which projects out or rearranges dimensions. Rearrangement can be treated as a presentation issue and omitting a dimension can be interpreted as a special roll-up directly up to the coarsest granularity (denoted by ALL or $\emptyset$) of a dimension. Other operations are not considered as basic functions now.

3.3 Aggregates and Summarizability

Summarizability of aggregations [58] is an important property in statistical and multidimensional databases. It means an aggregation value for a superset can be computed from the aggregation values of its subsets that form a partition.

Three basic types of aggregation exists [59]:

Distributive: If the aggregation values of disjoint sets are given, the aggregation value of their disjoint union can be computed using an aggregation operation. The *number of items* or *the sum* of some property of items are distributive aggregations with the SUM operation.

Algebraic: The aggregation value for any set can be computed by a formula of other, distributive aggregation values of the same set. A *mean* value is an algebraic aggregation, which can be computed by division. To reach summarizability, the sum and the number of items must be stored as auxiliary data.

Holistic: The aggregation value of a disjoint union cannot be computed based on aggregation of its subsets. The item level data must be recalled. The *median* is an example of holistic aggregation.

A distributive aggregation is easily summarizable. An algebraic aggregation needs some extra aggregation data and a holistic aggregation needs item-level data. Certainly, some aggregations (e.g. the third largest value) lie on the border between algebraic and holistic aggregation. The concept of algebraic aggregation can be extended into this direction [59].

In fact, the concept of summarizability is closely related to semantics. If an aggregation is possible along some dimensions and not possible along others, it is usually referred as *semi-summarizable or semi-additive* [50]. For instance, the number of inventory items is a distributive aggregation and can easily be summarized if we have several warehouses. However, a similar temporal summarization produces invalid results if it is based on snapshots (inventory at different time points). It does not take into account that an item exists in a time interval and the same item can be counted twice. The semantically right interpretation of this aggregation is summarizing the number of *item-time* pairs (items at a certain time) and not just *items*. Summarizability of item-time pairs along other dimensions than time is valid as the sum of items because time remains constant during such aggregations.

Type of an aggregation attribute with respect to temporal summarization can be one of the following [58]:

Flow: A flow type aggregation records values of events in certain time points, cumulated for certain periods of time (e.g. the number or the value of transactions in a month).

Stock: A stock type aggregation records the state in specific points of time by values cumulated for items (e.g. inventory records: number of items or the sum of prices of items as a monthly record).

Value/unit: Similarly to the stock type it is recorded in points of time, but semantically refers to a property of a single item or an item type, not cumulated (summarized) for items (e.g. the price of a certain item or an exchange rate of a currency).

Summarizability depends on the aggregation operation as well. An average over items or time is valid for all the above three types. In fact, non-temporal summarizability using the sum operation is semantically meaningful for flow and stock types. A similar temporal summarizability is valid for flow type aggregations only (cumulation).[2]

Spatial summarizability over geographical fields is rarely studied. In fact, it can face similar problems as temporal summarizability, due to the continuous nature of geographical space (e.g. counting lakes by countries and then summarizing them causes shared lakes counted twice or more times). Temperature, rainfall or the number of accidents in a region yield spatio-temporal aggregation tasks and may extend the above types.

3.4 Motivating Cases

This section presents some cases that need a more sophisticated schema description and navigation framework, which can be based on a general data model of partitions. Focus is on multidimensional data but other issues are also considered.

In some cases, the structure of the available data does not fit in a clean dimensional hierarchical schema and so the classical multidimensional model should be generalized. In such cases an approximate schema is designed, which has usually inherent redundancy and many null values with or without explicitly declaring structural nulls or constraints among dimensional attributes. All this may cause poor navigation facilities, unreasonable aggregations and comparisons [60, 59].

The vision of partition database model is to establish a well-founded formal framework based on the traditional relational model, which copes with various needs as illustrated below, by generalizing the multidimensional schema and allowing to declare more sophisticated structures or relationships. The aim is to take maximal advantage of aggregated data and possibly partial information that is available and perform computations using pre-aggregates, without using fine granularity raw data. While a couple of formalizations exists for data cubes and dimension hierarchies [76, 3], operations usually based on a uniform finest granularity (fact table) and do not consider dealing with heterogenity or missing information in a data cube schema.

The following subsections give an overview of the cases which may arise and are tried to be handled by the partition framework.

3.4.1 Partially applicable attributes

Some of the dimensions or dimensional attributes may not be applicable to all data. Some data may not be available in all granularities of a dimension. Such cases lead to structural nulls. The partition framework tries to model these by making explicit what attributes are applicable to which subsets of the data. With this approach, the schema reflects the structural constraint and a drill-down operation can only be applied when the user works with an appropriate slice of the data cube. The following cases can lead to this situation, for instance:

[2]The term *cumulative* aggregation attribute refers to a distributive, flow type aggregation value for sets throughout the thesis (see also Section 4.1).

Nonapplicable classification. Recall Example 4 from Section 3.2. Travelcards may further be classified according to their subtypes (validity: day, week, month, year) but this classification attribute does not apply for single tickets. Another case is when the big cities are divided into districts while the small ones are not (Example 3).

Extensive is-a hierarchies. If a dimension refers to a part of the schema that consists of an extensive is-a hierarchy, many dimensional attributes become applicable to subsets of the data only. [57] defines *weak attributes* for this case and introduces normal forms that treat them as *property attributes* instead of *category attributes*. They do not participate in dimensional hierarchy and it leads to the flattening of the hierarchy. However, it is reasonable for extensive is-a hierarchies to keep the structure and to allow drill-down navigation for cube slices, through several levels of the hierarchy. In Example 4 the vehicles may have such an extensive hierarchy (bus type: electric or not, if not electric: fuel consumption; train type: underground, light rail, tram, funicular, etc.; nr. of cars for specific train types; gauge; ...).

Extra details. In some situations, the database may contain extra details for a specific subset (slice) of the cube. We may have a street level granularity for a specific city (study area) while data of other cities are available only on district level. They can be used for comparison with the aggregated data of the study area. A drill-down is applicable to the study area that reveal details cannot be obtained outside the study area. A typical case is when the user has detailed data, creates an own dataset (myDB) and relates it to the coarser, global database.

Incomplete data. It is basically similar to the previous case but the reason is different. [44] argues that derived data products will have great importance in order to handle the data avalanche the eScience is currently facing. Historical data is archived or deleted and only some aggregates remain for surveys, statistics, comparisons. Another case is when a part of the database may not be available due to storage, security or other reasons. In the postage example, the street-level details of the number of deliveries for some cities may be unknown or the user may have insufficient privileges. It is reasonable to have a unified view over what is available. The point of [69] is adopted by assuming that the available information is based on subsets determined by classification hierarchies and can always be refined if we get data of finer granularity.

3.4.2 Partially applicable attribute combinations

Suppose the dimensional attributes are applicable to all data but we have insufficient information to create the data cube in full detail due to missing dimension (attribute) combinations. It is important to declare the known information exactly in order to provide proper partial navigation facilities that allow exploring all available dimension and granularity combinations. Although such situations can usually be solved by splitting the data cube, hiding the split and reflecting the structure in an unified schema seems a better, more user-friendly solution for the navigation operations and gives the facility to logically derive some of the missing data in the background, using metadata and database constraints or to recognize from the structure which data is missing and to input the missing data into the database.

This can be viewed as an extension of the above case. Therefore, only two examples are given.

Historical data. As mentioned above, historical data may not be available in full detail any more. It meant in the above case that some slices of the data cube cannot be unfolded by drill-down in full detail because one or more attributes are not applicable for universal refinement. There can be situations when all attributes are applicable in full detail but the data cube cannot be constructed in full granularity. Recall Example 4 and suppose that journey data of the public transport company is kept aggregated by $(Ticket, Line, Time)$ and $(Ticket, Vehicle, Time)$ up to last year (assuming only these dimensions for the sake of simplicity now). This two cubes are sufficient to survey the traffic on lines (services) and the vehicle loads. However, it is impossible to construct the cube by $(Ticket, Vehicle, Line, Time)$ even if we had a $(Vehicle, Line, Time)$ cube as well. The two historical cubes can be put into a unified schema with the full detailed cube of recent journeys, provided with well-defined restricted navigation operations.

Data integration. Several papers consider unifying different data cubes by dimension matching and similar techniques (e.g. [15]). Without considering the integration process itself, a means of describing the resulting schema is aimed. It is often the case that the dimension matching cannot be vica-versa and we either loose information while making the integrated schema by dropping those levels that cannot be matched to the schema of the other cube, or we get partially applicable attribute combinations. The resulting situation is basically the same as with historical data above, except when aggregation inconsistency occurs (see Inhomogenity of Aggregation later in this section). If the source of the information for the integrated cubes is the same, the aggregation values will be consistent (e.g. extracted from the same operational database or stored as different parts of the same distributed database). Data integration with dimension matching also allows derivation due to functional dependencies among attributes [69].

3.4.3 Complex dimensional structures

Dimensions do not always have a structure forming a simple hierarchical chain. This leads to alternative drill-down/roll-up navigation paths, nested dimensions or extra dependencies between dimensional attributes. For some parts of the cube other hierarchical structure may apply compared to others. A unified hierarchy can be given on the global level but navigation on slices can take advantage of structures valid for the slice only. These structural constraints are also important for detecting data errors and providing consistent extension of the database, as well as help by comparisons of different parts of the data cube. If the data is not available in full detail, constraints allow derivation of what can be calculated from the known information as mentioned already.

Alternative hierarchies. A classical example for an alternative hierarchy is when both *week* and *month* is present in a *time* dimension. Starting from *day* level, it leads to two different paths of roll-up navigation since weeks and months are not refinements of one another. This case is considered as almost standard [15, 47] and needs to be modeled, certainly.

Inhomogenous hierarchies or structures. Sometimes a dimensional hierarchy is not homogenous for the whole dataset but it may worth declaring different hierarchies for subsets. Division policy for zones of ZIP codes may differ for different cities, whether they have city districts, and whether the district attribute is part of the hierarchical chain or forms an alternative path as in the previous case. If we want to focus on precise description of the data structure, it is important to declare on schema level which subsets (cities) have which policy, in order to define proper selective navigation paths. Combined with the case of partially applicable attributes, it can lead to heterogenous schemata. For instance, the administrative division and the number of provincial levels (state, county, region, area, district, etc.) may differ from country to country as illustrated on Figure 3.4. In the traditional multidimensional modeling, we have again either structural nulls or loss of information (omitting a province level). If the schema wants to capture the exact structure, different hierarchies must be defined for each country and selective navigation must be provided as well as ad-hoc correspondence among dimensional granularities of different countries. For instance, second-level provinces of one country can be treated together with

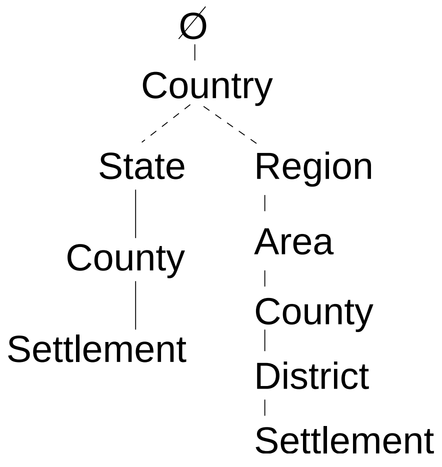


Figure 3.4: Heterogenous hierarchy for a location dimension, due to differences of administrative systems in specific countries

the third-level provinces of another country for particular statistical purposes. Due to extreme size differences, provinces of a country may be compared with whole countries. This may even vary from task to task, therefore, a fixed matching on schema level is not always suitable. Combined with incomplete data, these structural descriptions also help identifying the subsets where the information in the finest granularity can be derived from the known information and the subsets where not.

Interrelated dimensions. In the transport example, the Line (service) number determines the vehicle type of some granularity, e.g. a bus line cannot be served by underground trains. This shows that the dimensions are not independent of each other and there are large, well-defined regions of the multidimensional space that contain no values (structural nulls). The simplest case will be considered, if a functional dependency holds between two dimensional attributes.

Nested dimensions. If an alternative dimensional hierarchy is present with long separate paths, there may be a need for not choosing one of them for a navigation, but look at both, simultaneously. It leads to the separation of one dimension to two, for those granularities that participate in the alternative part of the hierarchy. If both month and week is present for the time dimension, it may be needed to split and treat them as two dimensions (sub-dimensions) for a specific operation.

3.4.4 Inhomogenity of aggregation

Of course, all the above cases may be combined with the availability, derivability, summarizability of aggregation values. Complicated cases of aggregation inconsistency or partial availability will not be considered deeply yet, since the main focus is on the basic structure of subsets and it adds extra complexity for the first steps. A simple case will be assumed: a summarizable, well-defined (distributive, flow type) aggregation value is stored for each known granularity combination in the data cube. Of course, the goal will be to properly handle such aggregation issues in a further phase of the development of the model.

3.4.5 Distributed storage

If a huge database is distributed over a network in a peer-to-peer fashion, it is important to know what exactly is available currently on a specific node. A data cube may only be available partially, moreover, some missing data may be computed from the available data without the need of retrieving other parts of the database from another node. If a constraint applies to a specific part of the database, which is the part actually queried, there can be much less need to copy large datasets over the network since missing information is determined by the constraint, and only the necessary extension must be retrieved.

3.4.6 Partitioning in general. Special applications

The concept of partitioning appears in special applications such as GIS (Section 3.1). Some of the topological relationships may be described in terms of partition combinations. It becomes more important if we think about the wide variety of representations of the sets and the computation costs of operations in these representations. If the structure of the sets is modeled and stored in a simple relational database with some aggregation data, many operations can be performed without the need to use high-cost item-level data. Furthermore, in some cases, the data of the highest granularity can – at least theoretically – always be refined further, but it is certainly not needed for the whole dataset. This introduces extra heterogenity in addition to the variety of representations and strengthens the need for a unified schema and navigation framework on a higher level.

3.4.7 Metadata in general

The importance of metadata is raising in general. As the amount of data assets (files, database objects, etc.) becomes higher and higher, more sophisticated metadata handling is necessary in order to find the searched data and navigate through the amount of data assets. [43] points out the need for set-based metadata handling and some of the problems the conventional directory-tree-based file systems currently facing. Some key points are similar to the topic of partition framework, since different values of a metadata field partitions the data assets and it is a reasonable user need to overlay these partitions and navigate through the available data in a slice/roll-up/drill-down fashion. It is supposed that several of the issues discussed above must be coped with for a proper partition-based metadata database framework.

3.5 Towards a General Partition Database Model

We are facing the kernel problem of designing the structure of a multidimensional-like database with several heterogenous relations, possibly with different granularities and dimensions. The contribution is based on the following aspects:

Generally, multidimensional data may be viewed as a multiple partition over a population. Elements are partitioned according to each attribute contributing to any dimension (different granularities of each dimension are treated as partitioning attributes). The structure of partitioning is based on the classifying attributes and the dependencies among them. It is assumed that the population-level information is not given, just aggregates of subsets defined by the combination of partitioning attributes in the finest possible granularity. In principle, each subset definable by attribute selection conditions can be divided further by new attributes adding a new relation into the system. Partitioning implies disjointness which is a key to summarizability. A database may contain a couple of relations but the structure is based on the classifying attributes if their consistency is ensured. An attribute has its base set (population) to which it is applicable.

This allows hiding of the relation names from the user who can perform operations in a unified way, by using the attribute structure. Any decomposition (including redundant) can be chosen for the data considering efficiency, without affecting the user level based on dimensioning attributes. This approach is similar to the universal relation assumption (URA, [2]) with the difference that the universal relation cannot always be constructed in full detail from the available data, but theoretically exists and depends on the (external) item-level data.[3]

Spyratos ([71, 70, 25, 52], see also [55, 53, 54, 56]) introduced the concept of partition database in a deductive and functional approach. In that terminology, an attribute is an atomic partitioning: a mapping from objects to identifiers representing categories. It has a reasoning facility so that one can derive information from the given data, taking into account relationships among partitions of the same base set. The relational representation of the proposed partition model in this thesis (Chapter 4) is based on the conjunctive part of that model, extended with the concept of cumulative (aggregation) attributes, different base sets and extra integrity constraints (base set constraints, which are not only constraints but provide metadata of partition relations). Applying the well-known theory and tools of relational databases with special semantics provides a formalism to describe structural and statistical information on multiple, simultaneous partitionings without having the full knowledge on the partitioned elements. A partition relation describes nonempty intersections of subsets. It can also be viewed as an aggregated fact table of a star schema if the dimensions are treated as partitioning attributes. Functional dependencies (conjunctive dependencies, see [70]) denote hierarchies and more complex relationships among the attributes. A simple partition algebra will be defined for partition relation operations. If a query can be formulated in this algebra, there is no need to 'drill down' to the population-level data to answer. It is especially important for special applications and representations like GIS data since no costly geometric operations are needed for such queries. Base set constraints determine the validity of operations. They prescribe syntactical conditions on the relations themselves and hold extra information (metadata) about the relationships of base sets being partitioned.

Based on this relational representation a higher-level view can be defined as a unified description and navigation framework for a partition database (Chapters 5 and 6). On this, *structural level*, relation names do not appear and a set- and attribute-oriented formalism provides the database interface. A graph of attribute structure is used as the basis of navigation where each node corresponds to a cuboid (known partitioning). It is a generalization of the lattice of cuboids for a conventional data cube as well as the generalization of the semilattice graph of closed attribute sets w.r.t. the set of functional dependencies (Section 2.3.4). If such a structural graph is given, any decomposition (star, snowflake, 3NF, materialized views) can be used for the relational representation ([63] considers the arguments related to correctness, effectiveness and usability). According to Spyratos and the relational normalization theory extended by some specific issues, a semi-automatic method is available to find an appropriate partition relational schema for the described set structure schema (see Section 5.3). The queryable partition combinations form a special view system.

The partition model allows handling of several datasets in a unified, 'seamless' way. Data access can be based on the conceptual model, i.e. what the data refer to and not where it is stored (in which dataset). A base set can be defined by a disjoint union of two selections, e.g. the union of slices of two different data cubes. Such new data can be divided further by importing a more detailed dataset for that base set.

A data cube can be modeled as a special case. The partition model treats different resolutions of the same dimension the same way as two different dimensions in a multidimensional schema. This allows more flexibility for operations as well as for defining relationships among different resolutions of a dimension or even between dimensions, especially if the dimensions have alternative hierarchies or more complex structures. It provides a convenient representation if the data cube is sparse: the corresponding partition relation will contain the attribute value combination for each nonempty cell as a tuple. The concept of dimension may

[3]The theoretical universal relation contains the raw, population-level data.

become implicit or relative, depending on the independence of attributes. Furthermore, if the data cube has some parts (subcubes) which can be partitioned further according to some specific (local) aspect, it can be captured by a nested cube: introducing another partition relation, only for the specified part. Integrity can be ensured by semantical constraints.

The outline of the next part is as follows: Chapter 4 introduces the partition model (based on Spyratos' work) with basic definitions and briefly illustrates the traps of a naive usage of algebra operations, then presents database constraints and proper partition algebra operations. Chapter 5 introduces the structural graph notation as a generalization of the lattice of cuboids with its possible extension. It provides a basis for possible navigation operations. Chapter 6 puts the partition model into a four-level framework, demonstrates basic cases that can be handled and closes with evaluating and future issues (basic partition operations can be found in Appendix A). Starting with Chapter 7, handling of functional dependency sets is studied. It is a more general topic but can be used in the partition model by describing the refinement structure of partition attributes.

Part II

A General Partition Data Model

Chapter 4

Simple Partition Data Model

This chapter introduces the concept of partition relation, based on the reformulation of the model presented in [52, 71], extended with the notion of cumulative attributes. It shows how algebra operations can be interpreted over partition relations and introduces new types of integrity constraints (base set constraints) that ensure the valid operations between them. These concepts allow the traditional relational data model to be used with partition semantics. More sophisticated data organization for a partition database will be considered in subsequent chapters.

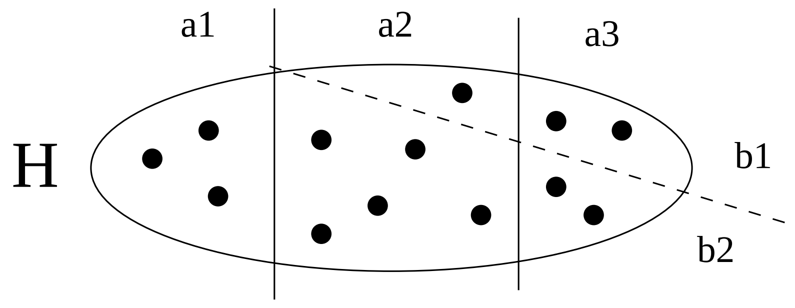
4.1 Partition Semantics of Relations and Cumulative Attributes

Traditional relational algebra queries are generic because symbols used in relations are independent of the denoted objects and a permutation of names keeps query results invariant.[1] Constants in the database have no properties other than the relationships with each other specified by the database [2]. Interpreting a relation involves assignment of semantics to attributes and the symbols occuring in its tuples. By getting rid of the additional structural information of symbols (eg. order of values) we get a simple, kernel model.

In this partition model, an n-ary relation instance represents an n-way partitioning of the base set by stating which sets of different partitionings intersect. Each nonempty intersection appears as a tuple in the relation. A tuple contains names of sets whose intersection is nonempty, optionally extended with some cumulative attributes of the intersection. The naming convention of sets is not resolved inside the database, it is considered as external information. Interpreting a partition relation involves assignment of semantics to attributes and mapping of symbols occuring in tuples to sets of objects in the underlying population.

Another syntax for the partition model is introduced as of [52, 71] extended with cumulative attributes. Focus is on the operations that can be performed without knowing the particular semantics (the population and how the items are exactly partitioned) since partition semantics is not contained in the database. Some operations may only be valid if we have some (external) information on the possible semantics, eg. the base sets of two relations are equal. Such information can be formalized using semantical integrity constraints. Some basic operations will be introduced, which are cumulation preserving and closed under partition relations, in a relational algebra fashion. The model may be extended by other operations and other types of schemata as well.

[1]Permutation of the symbols is commutative with query functions except selections with constants in their condition.



R:

A	B	N
a1	b2	3
a2	b1	1
a2	b2	5
a3	b1	2
a3	b2	2

Figure 4.1: A simple example of a partition relation with a cumulative attribute (see Example 5)

4.1.1 Partition Relations and Semantics

Let us consider a simple example first, illustrated on Figure 4.1:

Example 5. *Assume $\mathcal{H}$ is a population of items partitioned in two different ways. Let partition A consist of disjoint subsets a_i ($i \in \{1, 2, ..., k\}$) and B of b_j ($j \in \{1, 2, ..., l\}$), both covering $\mathcal{H}$. Combining these two partitions we get a finer partitioning out of pairs (a_i,b_j) having nonempty intersections. Intuitively, a* partition relation *R of h w.r.t. partitions A, B has A and B as its attributes and the set of tuples $\{(a_i, b_j) \mid a_i \cap b_j \neq \emptyset\}$. If an aggregation value N is known for each of these subsets calculated by function f over sets (eg. the number of elements or the sum of a property over the elements of the subset), it may be attached to the partition relation as an extra* cumulative *attribute: $R : (A, B; N)$, $R = \{(a, b, n)|a \cap b \neq \emptyset, n = f(a \cap b)\}$.*

The basic definitions follow. A partitioning attribute system is basically a relational attribute system. Attributes correspond to partitions. A partition relation is a relation of a partition schema. It desrcibes a multiway partition (overlay) of a base set. Symbols used in the relation denotes subset names. Partition semantics relates these names to the population data, i.e. it resolves set names in a consistent way. Partition semantics itself is not considered to be part of a partition database.

Definition 1. Partitioning attribute system:
Let $\mathcal{A}$ be a countable set of symbols denoting sets, $Attr$ be a countable set of attribute symbols and $DOM : Attr \to 2^{\mathcal{A}}$ be a function determining the type (domain) of each attribute. In such a case, $(Attr, \mathcal{A}, DOM)$ is called a partitioning attribute system*.*

Definition 2. Partition schema & relation:
A relational schema $\mathcal{R}$ is an n-way partition schema*, if it has n attributes taken from a partitioning attribute system. We call an instance of a partition schema a* partition relation*.*

Definition 3. Partition semantics:
Let R be a partition relation of schema $(A_1, \ldots, A_n)$ and $\mathcal{U}$ a nonempty set, called the universe of discourse[2]*. A possible* partition semantics *of R over $\mathcal{U}$ is the following:*

$$\Phi = (\mathcal{H}; \varphi_1, \ldots, \varphi_n)$$

where

[2]Please note that universe refers to the population in this terminology and not the set of partitioning attributes as in eg. [70, 54].

1. *$\mathcal{H} \subseteq \mathcal{U}$ (called the base set of R), $\mathcal{H} = \emptyset \Leftrightarrow R = \emptyset$,*

2. *$\forall i \in [1 \ldots n] : \varphi_i : \pi_{A_i}(R) \to 2^{\mathcal{U}}$, i.e. a subset of $\mathcal{U}$ (may extend beyond $\mathcal{H}$) corresponds to each symbol of $\mathcal{A}$ occuring in R,*

3. *$\forall i \in [1 \ldots n] : \bigcup_{a \in \pi_{A_i(R)}} \varphi_i(a) \cap \mathcal{H} = \mathcal{H}$ and $a_1 \neq a_2 \Rightarrow \varphi_i(a_1) \cap \varphi_i(a_2) = \emptyset$ (i.e. the partition according to A_i determined by φ_i covers $\mathcal{H}$ and has no overlapping component sets),*

4. *$\forall (a_1, \ldots, a_n) \in R : \bigcap_{i \in [1 \ldots n]} \varphi_i(a_i) \cap \mathcal{H} \neq \emptyset$ (each tuple of R represents a nonempty intersection of sets). For the sake of simplicity, a notation is introduced: $\varphi_\cap(a_1, \ldots, a_n) \overset{def}{=} \bigcap_{i \in [1 \ldots n]} \varphi_i(a_i)$.*

A partition relation with an assigned partition semantics describes a multiway partitioning of the base set $\mathcal{H}$. Tuples of the relation correspond to set intersections of different partitions which are nonempty. The definition implies $\bigcup_{t \in R} \varphi_\cap(t) \cap \mathcal{H} = \mathcal{H}$ (i.e. the partition system covers $\mathcal{H}$) and $\varphi_i(a_i) \cap \mathcal{H} \neq \emptyset$ for each possible i and a_i (empty sets are not allowed to be assigned to symbols). To simplify the notation, function symbols φ_i of the semantics may be omitted, i.e. a set $\varphi_i(a)$ will be denoted by the symbol a itself where it causes no confusion.

The role of $\mathcal{H}$ may be seen as redundant. If only one partition relation is considered, then the semantics should be chosen so that the union of intersections represented by tuples equals the base set. However, if two or more relations are considered together as a database, their base sets do not need to be equal. Symbols are aimed to be global for a specific attribute and restriction to base sets is performed by intersection. Therefore, a missing tuple does not imply the corresponding set is empty, it may only be outside of the scope (base set) of the relation.

The following definition refers to compatibility of semantics. If two relations have an attribute in common, it does not mean automatically that the are semantically related. Compatibility ensures their relationship and states that the projection of the two relations on the common attributes are consistent, i.e. they partition the intersection of the two base sets the same way. It can be assumed that they come from the same source: a partition over the union of their base sets.

Definition 4. Compatibility of semantics:
Let $R : (A_1, \ldots, A_n, B_1, \ldots, B_k)$ and $S : (B_1, \ldots, B_k, C_1, \ldots C_m)$ be two partition relations with semantics $\Phi = (\mathcal{H}; \varphi_1, \ldots, \varphi_n, \chi_1, \ldots, \chi_k)$ and $\Psi = (\mathcal{I}; \chi'_1, \ldots, \chi'_k, \psi_1, \ldots, \psi_m)$, respectively. Note that in this case, $\pi_{B_i(S)}$ is the domain of χ'_i and $\pi_{B_i(R)}$ is the domain of χ_i; let $B_{\cap i}$ denote their intersection $\pi_{B_i(R)} \cap \pi_{B_i(S)}$. Φ and Ψ are compatible with each other *if*

$$\forall i \in [1 \ldots k] : \chi_i|_{B_{\cap i}} = \chi'_i|_{B_{\cap i}}$$

i.e. the common symbols of the two relations refer to the same sets.

Compatibility means a partitioning on $\mathcal{H} \cup \mathcal{I}$ by $(B_1, \ldots, B_k)$ can be correctly constructed by union. However, cumulative attributes cannot be kept since compatibility does not mean the same rows correspond the same sets (the base sets may be different). More strict concepts of compatibility will be formalized as semantical constraints in Section 4.3 (see also Figure 4.2).

4.1.2 Cumulative Attributes and Simple Partition Database

Partition relations can describe facts about the nonemptyness of intersections only. The notation can be extended to keep track of cumulative attributes (eg. some statistics) of these intersections.

Definition 5. Cumulative attribute system:
Let $DAttr$ be a countable set of attribute symbols and $DDOM$ be a function that assigns a domain to each of them. Furthermore, let DOP be a function that assigns an associative, commutative operation to each of the attribute symbols on its domain. Using these notations the triple $(DAttr, DDOM, DOP)$ is called a cumulative attribute system.

Definition 6. Partition schema & relation with cumulative attributes:
An $\mathcal{R}$ relational schema is a partition schema with cumulative attributes*, if its attributes form two groups as $\mathcal{R} = (A_1, \ldots, A_n; X_1, \ldots X_s)$ where A_i's form a partition schema and X_i's are taken from a cumulative attribute system. A* partition relation with cumulative attributes *is a relational instance of $\mathcal{R}$ obeying the functional dependency $A_1 \ldots A_n \rightarrow X_1 \ldots X_s$.*

Partition semantics can be attached to instances with cumulative attributes by considering the partitioning attributes only. Cumulative attributes do not need extra semantics formally since the data they hold refers to the intersection of sets assigned to the symbols of the tuple. They are automatically adjusted 'in the background' by relational operations on partitioning attributes.

It is important to note that only summarizable aggregations are considered now, in order to be able to compute the value for a union using the values for the unified sets. They must distributive and – if time appears as a partition – flow-type as defined in 3.3. COUNT, SUM (both with operation '+'), MIN, MAX are examples of possible aggregation operators that can be simulated by this model. Extension to other types (eg. algebraic, holistic) is a promising future issue. An algebraic aggregation like AVG must be generated explicitly by its distributive components (SUM/COUNT).

Based on all these, the definition of simple partition database follows:

Definition 7. Simple Partition database schema, instance & semantics:
A simple partition database schema *$\mathcal{P}$ is a set of partition schemata over a partitioning attribute system (possibly with cumulative attributes). A* partition database instance *is a relational database instance of $\mathcal{P}$. A* partition semantics of a partition database *is a set of partition semantics assigned to each of its relations such that they are all compatible with each other.*

4.2 The Naive Algebra and its Problems

Relational algebra operations can be applied to partition relations. However, there is no guarantee that the traditional operations yield partition relations and the semantics of the result can be meaningfully constructed from the semantics of the operands. Partition selection and projection must be defined as follows:

Selection is defined in the natural, traditional way: tuples of the operand relation is kept which obey the selection condition and form the resulting relation. This yields a partition relation over a smaller base set.

Projection, however, must be defined slightly different since projecting out a partitioning attribute can lead to duplicate tuples with the same values of remaining partitioning attributes. In order to ensure that the result is a valid partition relation, the operation must unify these and summarize the aggregation values, using the default aggregation operator given in the attribute system.

4.2.1 Selection-Projection Problem

Assume two partition relations $R' : (A, B; N)$ and $S' : (B, C; N)$. Let $R = \pi_B(\sigma_{A=a}(R'))$ and $S = \pi_B(\sigma_{C=c}(S'))$ and suppose $b_2 \in R, S$. In this case, nothing guarantees that b_2 denotes the same set in R as in S even if it was so in R' and S' and the semantics are compatible. Computing $R \cup S$ or $R \cap S$, for instance, is impossible inside the partition database, we need to know the exact semantics. Figure 4.2 in the next section illustrates three different cases. If we knew exactly what sets the relations refer to (especially $b2$), we could compute both operations but our aim is to use what we know on the partition level about the subsets only. Therefore, the operations cannot be always computed. We need more strict concepts than compatibility of semantics.

4.2.2 The Join Problem

Considering join, to preserve closure of the operations, the result must be a partition relation. This can not be guaranteed in the general case (eg. joining the two relations of Example 11 (see later) would not yield a 3-way partitioning, just an upper bound of it [52, 25]). Even if the join yields the correct partition structure (i.e. a proper join dependency holds), the cumulative values cannot be computed from the operands.

4.3 Integrity Constraints

4.3.1 Functional Dependencies in Partition Schemata

Some projections of a partition relation may be equivalent to each other as they represent the same sets, i.e. if the attributes projected out do not divide any of the sets further. At the design phase, functional dependencies [2, 33] can be used to declare necessary equivalences. In general, partitioning according to the left-hand side of a dependency refines the partitioning defined by the right side. [70] introduced the concept of *conjunctive dependency* stating such an equivalence and studied its close connection to functional dependencies.

Example 6. *Recall Example 3. Focusing on an address dimension and aggregating (dicing) over all others, let* $R : Post = (S, Z, D)$ *represent a three-way partitioning of a city (*S *stands for streets,* Z *for ZIP codes and* D *for districts). Two-way and simple partitionings (eg. by SZ, S, Z) can be derived by projection. If* $Z \rightarrow D$ *holds then partitioning by* ZD *is essentially the same as partitioning by* Z *since they both represent the same sets (regardless of what elements the sets contain), and the same holds for* SZD *and* SZ*. Including* D *affects the set names only and adds an aggregation possibility.*

Definition 8. Equivalence of partitionings:
Given a partition schema $\mathcal{R}$ *and a set of functional dependencies* $\mathcal{F}$*, we treat two attribute sets* $\mathcal{A}$ *and* $\mathcal{B}$ *as* equivalent *w.r.t. partitioning if and only if* $\mathcal{A}^{+(\mathcal{F})} = \mathcal{B}^{+(\mathcal{F})}$ *(their closures are equal).*

Closed attribute sets can denote equivalence classes of partitionings:

Proposition 1. *Given an instance* R *of* $\mathcal{R}$ *satisfying a set* $\mathcal{F}$ *of functional dependencies, there is a one-to-one correspondence between the closed attribute sets w.r.t.* $\mathcal{F}$ *and the possible different partitionings resulting by projections of* R*.*

More on dependencies and partition equivalence in terms of structural graphs will be discussed in Chapter 5.

4.3.2 Base Set Constraints

Base set constraints provide a way to put some extra, external information (background knowledge, meta-data) about the partitioning into the schema, such as two relations partition the same set or one is a subset of another. This knowledge is important in interpreting a partition database since it does not always follow from the data, and affects the validity of some operations.

A base set constraint constrains the properties of semantics that can be attached to partition relations. The constraints are all binary, i.e. refer to two relations. Let $R : (A, B)$ and $S : (B, C)$ denote these relations. Since a base set constraint refers to the partition semantics, satisfaction cannot be considered on the syntactical level, an interpretation (partition semantics) must be chosen for the database. Assume Φ_R, Φ_S are partition semantics of R, S, respectively. Satisfaction of a base set constraint ζ is denoted by $(R, \Phi_R), (S, \Phi_S) \vDash \zeta$. The satisfaction condition depends on the constraint type, but the common ground is that the two semantics must be compatible as in Definition 4.

For some constraints there exist pairs of relations R, S that cannot satisfy the base set constraint with any partition semantics (eg. if R contains different symbols than S in the same attribute column then they cannot partition the same set). Such a case should not occur in a database if a base set constraint is declared. This leads to the *syntactical aspect* of each base set constraint type: a condition for the partition relations R, S so that a valid semantics exists that obeys the constraint. The syntactical aspect is the database constraint in the classical sense. Some constraints have no syntactical part (the condition is 'always true'), i.e. there exists a valid interpretation for any two partition relations. The syntactical aspect of a base set constraint can be formulated as a theorem: it is a necessary and sufficient condition for the existence of valid semantics. If two partition relations obey the syntactical part of a constraint ζ, it is denoted as $R, S \vDash \exists\Phi\zeta$.

Base set constraints are presented in a tabular form (Table 4.1). Their particular types will be explained later in this section. Each row corresponds to a constraint type definition. Column ζ represents the notation of the constraint. Column $\vDash \zeta$ is the constraint definition: the condition for the attached semantics to be a valid interpretation. Column $\vDash \exists\Phi\zeta$ contains the syntactical aspect. Partition relations $R : (AB)$ and $S : (BC)$ are assumed (A, B, C denote pairwise disjoint attribute sets), with partition semantics Φ_R and Φ_S with base sets $\mathcal{H}_R, \mathcal{H}_S$, respectively. Strict versions of constraints are defined using an attribute subset $D \subseteq B$ (eg. $\theta(D)$ is a selection condition depending on D only). Definition of some constraints is based on a semijoin or selection that can be interpreted as usual. The base set of a semijoin or selection result is denoted by $\mathcal{H}$ with a proper subscript. Compatibility of semantics (see Section 4.1) is implicitly necessary for each of the semantical constraints, and compatibility itself can be treated as a semantical constraint too. We assume first that no cumulative attributes present in the schemata of participating partition relations.

Correctness of Table 4.1 is ensured by the following statement for each of the defined constraint types:

Proposition 2. *Given two relations R and S and a base set constraint ζ, there exist valid partition semantics, i.e. $\exists\Phi_R\Phi_S : (R, \Phi_R), (S, \Phi_S) \vDash \zeta$, if and only if R and S obey the syntactical aspect given in Table 4.1 for the type of ζ.*

If cumulative attributes are present, the constraints have extra semantics: they state cumulation consistency of the attributes with the same name. Eg. a base set equality constraint states that the sum (or other associated operation) of cumulative attributes are equal for the two relations and this holds for each row of the projections on the common attributes (see partition projection in more detail in Section 4.4).

Base set equality constraint

$\mathcal{H}_R = \mathcal{H}_S$ means the following: whenever the relations are interpreted with partition semantics Φ_R and Φ_S, respectively, their base sets must be equal. If $\pi_B(R) \neq \pi_B(S)$ then no valid semantics can be attached that obeys the base set equality constraint with the two relations. In fact, if $\pi_B(R) = \pi_B(S)$ then there exist semantics Φ_R and Φ_S such that they give a valid interpretation. Therefore, this syntactical condition is the syntactical aspect of this constraint type.

Constraint name	Notation (ζ)	Semantics ($\models \zeta$)	Syntactical aspect ($\models \exists\Phi\zeta$)
Base set equality	$\mathcal{H}_R = \mathcal{H}_S$	$\mathcal{H}_R = \mathcal{H}_S$	$\pi_B(R) = \pi_B(S)$
Base subset constraint	$\mathcal{H}_R \sqsubseteq \mathcal{H}_S$	$\mathcal{H}_R = \mathcal{H}_{S \ltimes R}$	$\pi_B(R) \subseteq \pi_B(S)$
Strict base subset w.r.t. D	$\mathcal{H}_R \sqsubseteq_D \mathcal{H}_S$	$\mathcal{H}_R = \mathcal{H}_{\pi_D(S) \ltimes R}$	$\pi_B(R) = \pi_B(S \ltimes \pi_D(R))$
Strict base subset w.r.t. $\theta(D)$	$\mathcal{H}_R \sqsubseteq_{\theta(D)} \mathcal{H}_S$	$\mathcal{H}_R = \mathcal{H}_{\sigma_{\theta(D)}(S)}$	$\pi_B(R) = \pi_B(\sigma_{\theta(D)}(S))$
Common sets constraint (strict compatibility)	$\mathcal{H}_R \sqcap \mathcal{H}_S$	$\mathcal{H}_{R \ltimes S} = \mathcal{H}_{S \ltimes R}$	always true
Strict common sets w.r.t. D	$\mathcal{H}_R \sqcap_D \mathcal{H}_S$	$\mathcal{H}_{\pi_D(R) \ltimes S} = \mathcal{H}_{\pi_D(S) \ltimes R}$	$\pi_B(R \ltimes \pi_D(S)) = \pi_B(S \ltimes \pi_D(R))$
Disjoint base sets	$\mathcal{H}_R \sqcup \mathcal{H}_S$	$\mathcal{H}_R \cap \mathcal{H}_S = \emptyset$	always true
Strict disjoint base sets w.r.t. D	$\mathcal{H}_R \sqcup_D \mathcal{H}_S$	$\pi_D(R) \cap \pi_D(S) = \emptyset$	$\pi_D(R) \cap \pi_D(S) = \emptyset$
Compatibility	$\mathcal{H}_R \parallel \mathcal{H}_S$	see Def. 4	always true

Table 4.1: Base set constraints with their semantics and syntactical aspects

Common sets and disjointness constraints.

A common sets constraint $\mathcal{H}_R \sqcap \mathcal{H}_S$ states that the common tuples of $\pi_B(R)$ and $\pi_B(S)$ refer to the same sets (their base sets must be chosen so). It is also called as strict compatibility since it enforces aggregation consistency of cumulative attributes on their intersection. If no common sets constraint holds, it is impossible to correctly compute the partitioning according to B with cumulative attributes on the intersection of the base sets of R and S because the same symbol in $\pi_B(R)$ may denote a different set as in $\pi_B(S)$. This difference is illustrated on the left part of Figure 4.2 (set $b2$).

A *disjoint base sets* constraint $\mathcal{H}_R \sqcup \mathcal{H}_S$ ensures the disjoint union of two partitions can be constructed (cumulative attributes of the two relations can be summarized to get the value for the union). The weak version does not imply any restriction on the syntax since the same symbols may refer to disjoint parts of the set they refer to. The lower right part of Figure 4.2 illustrates such an example. Note that $\mathcal{H}_R \sqcup \mathcal{H}_S \not\Longrightarrow \mathcal{H}_R \sqcup_B \mathcal{H}_S$ as opposed to strict versions of other constraints.

Example 7. *Recall Example 4 and assume the data cube Journey contains the current year's journeys in each time granularity $Year, Month, Day, Hour, Minute$. Aggregations of historical data for $Year, Month$, Day are stored separately in a cube $Journey_{old}$. To ensure consistency, a disjoint base set constraint $\mathcal{H}_{Journey} \sqcup_{Year} \mathcal{H}_{Journey_{old}}$ is declared on the schema level. This strict version states that the base sets can be separated according to values of $Year$, i.e. the same year can not occur in both of the datasets. The user level can present the two cubes as one data set for surveying, indicating that drilling down to $Hour$ and $Minute$ is restricted to this year's data.*

Base subset constraints

In the previous example, the dataset was split corresponding to disjoint underlying base sets. However, an alternative possibility is when the coarser data is kept for the whole base set due to efficiency reasons, together with the finer partitioning over a restricted base set. This is discussed next.

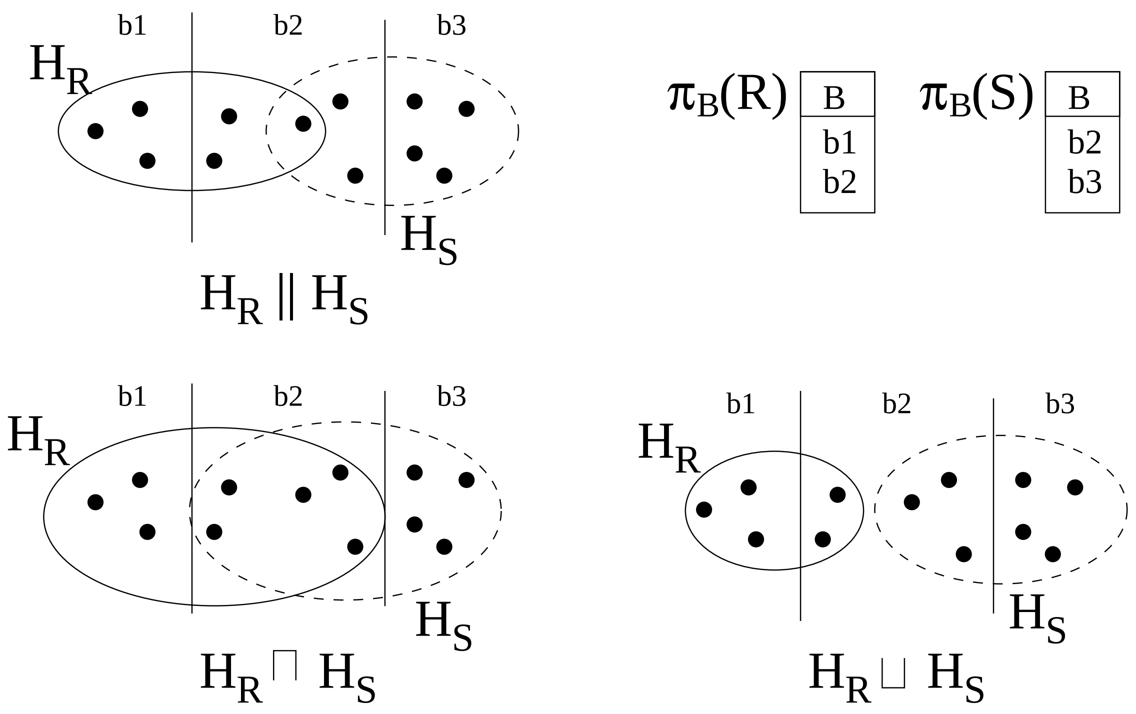


Figure 4.2: Possible interpretations of a database given compatibility (upper left), common sets (lower left) and disjointness (lower right) constraints. Since the database does not contain what $b2$ exactly mean in R and S, validity of union and intersection operations and the way how they are computed is determined by the base set constraints

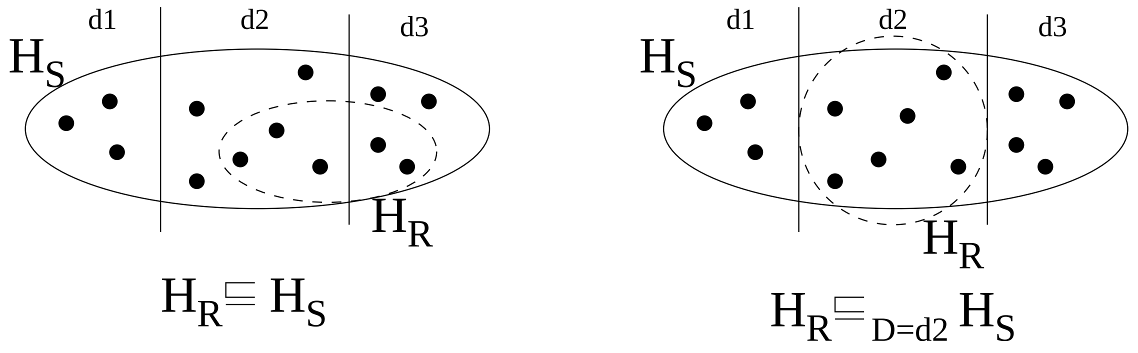


Figure 4.3: Base subset constraints

Nested (local, sub-) partitioning means refining a partitioning by extra aspects (attributes) only on some part of the base set. A base subset constraint $\mathcal{H}_R \sqsubseteq \mathcal{H}_S$ expresses that R describes a nested partitioning of $\pi_B(S)$. The strict version $\mathcal{H}_R \sqsubseteq_D \mathcal{H}_S$ of this constraint corresponds to the definablility of the base set of the nested partition by a selection criterion on S depending only on D. A more strict version of this constraint ($\mathcal{H}_R \sqsubseteq_{\theta(D)} \mathcal{H}_S$) prescribes the exact selection condition $\theta(D)$. Figure 4.3 illustrates a general and a strict case. A nested partition would mean H_R is partitioned further, by an extra attribute.

Example 8. *Let us extend the simplified Post schema of Example 6 to a whole country by an extra attribute C (City). Supposing only some bigger cities are divided into districts, the attribute D is not applicable for smaller places. Two partition relations Country : $\mathcal{C} = (S, Z, C)$ (for all the places) and BigCity : $\mathcal{B} = (S, Z, D, C)$ describe the partitioning. The constraint $\mathcal{H}_{\mathcal{B}} \sqsubseteq_C \mathcal{H}_{\mathcal{C}}$ ensures the base set of BigCity is definable by the attribute C of Country. This constrains the relations and determines applicable partition semantics.*

Relationships among constraints.

The basic relationships among the constraints are summarized below.

Proposition 3. *The following basic relationships hold (for all $D \subseteq B$):*

- $\mathcal{H}_R = \mathcal{H}_S \Longrightarrow \mathcal{H}_R \sqsubseteq \mathcal{H}_S \Longrightarrow \mathcal{H}_R \sqcap \mathcal{H}_S \Longrightarrow \mathcal{H}_R \parallel \mathcal{H}_S$
- $\mathcal{H}_R = \mathcal{H}_S \Longrightarrow \mathcal{H}_R \sqsubseteq_D \mathcal{H}_S \Longrightarrow \mathcal{H}_R \sqsubseteq_B \mathcal{H}_S \iff \mathcal{H}_R \sqsubseteq \mathcal{H}_S$
- $\mathcal{H}_R \sqsubseteq_D \mathcal{H}_S \iff \exists\theta(D) : \mathcal{H}_R \sqsubseteq_{\theta(D)} \mathcal{H}_S$
- $\mathcal{H}_R \sqsubseteq_D \mathcal{H}_S \Longrightarrow \mathcal{H}_R \sqcap_D \mathcal{H}_S \Longrightarrow \mathcal{H}_R \sqcap_B \mathcal{H}_S \iff \mathcal{H}_R \sqcap \mathcal{H}_S$
- $\mathcal{H}_R \sqcup_D \mathcal{H}_S \Longrightarrow \mathcal{H}_R \sqcup \mathcal{H}_S \Longrightarrow \mathcal{H}_R \parallel \mathcal{H}_S$

The general implication problem needs further research, but the form of constraints must probably be extended further.

4.4 Simple Partition Algebra

This section introduces some basic operations on partition schemata that are cumulation preserving and closed over partition relations, based on the named version of the monotone relational algebra (SPJRU, [2]). The operations based on partition attributes and automatically adjust cumulative attributes. They provide a way to construct partition semantics of the result if semantics of the operands are known. Operations are generic, they must not depend on the semantics. However, some operations can only be performed if some basic semantical information in the form of constraints is known. In some cases, this does not mean any syntactical restriction on the relations but knowing this kind of information ensures the result has a meaningful partition semantics (see also Section 4.2).

The complexity of this algebra is comparable to the Semijoin algebra [38], due to the restricted join operations. Since a functional dependency must hold for a valid join, the number of rows in the result is always bounded by the number of rows of one of the operands.

4.4.1 Unary Operations

Unary operations were already introduced informally in Section 4.2. They are formalized now.

In the following definitions, assume a partition relation with cumulative attributes $R : (A_1, \ldots, A_n; X_1, \ldots, X_s)$ and its partition semantics $\Phi = (\mathcal{H}; \varphi_1, \ldots, \varphi_n)$ over a universe $\mathcal{U}$.

Selection

Definition 9. Partition selection:
Let θ be a propositional selection condition with attribute-equals-constant style atomic conditions for partition attributes and arithmetic conditions for cumulative attributes. Relation $\sigma_\theta(R)$ is defined as in the traditional algebra and its partition semantics over $\mathcal{U}$ is defined as $\Phi|_\theta = (\mathcal{H}|_\theta; \varphi_1|_\theta, \ldots, \varphi_n|_\theta)$ where $\mathcal{H}|_\theta = \bigcup_{(a_1,\ldots,a_n)\in\sigma_\theta(R)}(a_1 \cap \cdots \cap a_n \cap \mathcal{H})$ and $\forall i \in [1 \ldots n] : \varphi_i|_\theta = \varphi_i|_{\pi_{A_i}(\sigma_\theta(R))}$.

The following proposition ensures the correctness of the definition and can easily be verified by referring to the definition of partition semantics.

Proposition 4. *$\Phi|_\theta$ is a valid partition semantics of $\sigma_\theta(R)$ over $\mathcal{U}$.*

Projection

Partition projection corresponds to set aggregation. It differs from traditional projection if the schema has cumulative attributes, since we must obtain a partition relation:

Definition 10. Partition projection:
Without loss of generality, assume $k \leq n$, $t \leq s$ and we project on attributes $A_1, \ldots, A_k; X_1, \ldots, X_t$. Partition relation $\pi_{A_1\ldots A_k;X_1\ldots X_t}(R)$ is defined as the usual $\pi_{A_1\ldots A_k}(R)$ extended with the cumulative attributes aggregated for each tuple using the aggregation operator assigned to their type. The partition semantics of the result over $\mathcal{U}$ is $\Phi^{A_1\ldots A_k} = (\mathcal{H}; \varphi_1, \ldots, \varphi_k)$.

Proposition 5. *$\Phi^{A_1\ldots A_k}$ is a partition semantics of $\pi_{A_1\ldots A_k;X_1\ldots X_t}(R)$ over $\mathcal{U}$.*

Renaming

Renaming of partition attributes is allowed only if the target attribute contains the domain of the source. Semantics of the result is obvious, however, further operations of renamed partition relations may require the compatibility of semantics. This must be ensured by the user since there is no facility of the simple partition database schema that keeps track of it. Renaming usually occurs in the context of disjoint union (see later in this section), which makes compatibility straightforward.

Renaming of cumulative attributes is possible if the target attribute has the same aggregation operator attached. The user must ensure aggregation compatibility with other relations and that the result has a meaningful semantics.

Renaming can be extended to apply a transformation on symbols of the partition relation tuple-by-tuple. The result remains a partition relation if the transformation is injective (otherwise it needs aggregation, similarly to partition projection). A transformation can be an arbitrary (computable) function or operator f that maps an n-ary tuple of given attributes (domains) to a single value of a given domain. This way, rename (with selection) corresponds to the *relabel* opeation in [39].

If a structure or algebra is assumed over the symbols, a relabelling transformation can be defined in terms of the given algebraic operations. As an example, if we assume the symbols as strings, the concatenation operation can be defined, denoted by the symbol '&'. The rename operation $\rho_{A_1 \& A_2 \to B}(R)$ results a partition relation of schema $(B, A_3, \ldots, A_n; X_1, \ldots, X_s)$. Actual semantics of symbols in B can be defined as the intersections of sets occuring in the semantics of A_1 and A_2 (without the empty intersections).

4.4.2 Binary Operations

For binary operations, two partition relations with cumulative attributes $R : (A, B; X, Y)$, $S : (B, C; Y, Z)$ are assumed with semantics $\Phi = (\mathcal{H};\ \varphi, \chi)$ and $\Psi = (\mathcal{I}; \chi', \psi)$, respectively, over the same universe $\mathcal{U}$. To construct the semantics of results, compatibility of Φ and Ψ must also be assumed, and some other dependencies may be needed. The attribute symbols may denote several attributes, eg. $A = (A_1, \ldots, A_n)$.

Operations semijoin, join, intersection and union are considered. One can easily verify the definition of the resulting partition semantics is correct in each of the following operations (like the propositions for the unary operations in the previous subsection).

Semijoin and Antisemijoin

Semijoin and antisemijoin can be performed in the natural way except that cumulative attributes originating from R are kept only, since it is interpreted as a special selection of R and the base sets may be different.

Definition 11. Semijoin and antisemijoin:
Relations $R \ltimes S$ and $R \bar{\ltimes} S$ are defined as the usual $R \ltimes \pi_{BC}(S)$ and $R \bar{\ltimes} \pi_{BC}(S)$, respectively. Their partition semantics over $\mathcal{U}$ are defined as $\Phi_{\ltimes S} = \Phi|_{\theta_{\ltimes S}}$ and $\Phi_{\bar{\ltimes} S} = \Phi|_{\theta_{\bar{\ltimes} S}}$, respectively, where $\theta_{\ltimes S} = \bigvee_{t \in \pi_B(S)} (B = t)$ and $\theta_{\bar{\ltimes} S} = \bigwedge_{t \in \pi_B(S)} (B \neq t)$.

Partition Join

Due to the join problem discussed in the previous section, restricted join operations are defined whose purpose is to derive the ways of partitioning which can be constructed from the known information. It is driven by functional dependencies:

Definition 12. Left-to-right partition join:
If the constraints $B \rightarrow C$ and $\mathcal{H}_R \parallel \mathcal{H}_S$ hold, the left-to-right join $R \overrightarrow{\bowtie} S$ *is a valid operation defined as the traditional join $R \bowtie \pi_{BC}(S)$. Its partition semantics is constructed similarly to a selection: $\Phi|_{\theta_{\ltimes S}}$, where the condition is $\theta_{\ltimes S} = \bigvee_{t \in \pi_B(S)}(B = t)$, but extended with ψ (with an appropriate restricted domain) corresponding to C.*

Right-to-left join *($\overleftarrow{\bowtie}$) can be defined symmetrically as $\pi_{AB}(R) \bowtie S$.*

A left-to-right join adds attributes of a coarser partitioning to R. A right-to-left join refines partitioning R by S. No essentially new partitioning is created, the result of $R \overleftarrow{\bowtie} S$ represents a subset of the set intersections of S (it corresponds to the same node in the structural graph as S). Cumulative attributes are kept only from the finer partition. For a left-to-right join or right-to-left join a common sets constraint $\mathcal{H}_R \sqcap \mathcal{H}_S$ is recommended but not necessary.

If both of the introduced versions of join can be performed on two relations and a common sets constraint holds, the following operation is valid:

Definition 13. Equivalence partition join:
If both functional dependencies $B \rightarrow C$ and $B \rightarrow A$ as well as the $\mathcal{H}_R \sqcap \mathcal{H}_S$ constraint hold, the equivalence join $R \overleftrightarrow{\bowtie} S$ *is a valid operation defined as the traditional join $R \bowtie S$ and its partition semantics is either the one of $R \overrightarrow{\bowtie} S$ or the one of $R \overleftarrow{\bowtie} S$ (they are equal in such a case), keeping the cumulative attributes of both relations.*

An extended join operation may be defined using the concept of horizontal decomposition [13]: if a functional dependency is not valid on the full relation, just on a part of it, a (partial) join may be computed. This join must be supported with a special semijoin operation, which retrieves the remaining part.

Intersection

Definition 14. Partition intersection (common sets intersection)[3] *is defined as a special case of equivalence join if $A = C = X = Z = \emptyset$ (and $\mathcal{H}_R \sqcap \mathcal{H}_S$ holds):*

$$R \Cap S \stackrel{def}{=} R \overleftrightarrow{\bowtie} S$$

4.4.3 Union

Two kinds of union operations can be defined, depending on the relationship of the two operands – whether their base set is disjoint or they have a known intersection by a common sets constraint. The actual base set constraint ensures the correct, consistent handling of the cumulative attributes.[4]

Definition 15. Disjoint partition union:
If $A = C = X = Z = \emptyset$ and the constraint $\mathcal{H}_R \sqcup \mathcal{H}_S$ holds, the disjoint union $R \uplus S$ *of partition relations is a valid operation defined as the traditional union $\pi_B(R) \cup \pi_B(S)$ extended with the cumulative attributes Y*

[3]A more general intersection operation ($R \cap S$) can be defined as a special case of semijoin. In that case, the name 'intersection' may be misleading since it does not build the intersections of sets described by tuples but keeps the tuples (sets) with set names occuring in both relations. Set difference ($R \setminus S$) can be defined similarly as a special case of antisemijoin, with the same warning.

[4]If no cumulative attributes present, the result of the two types of union is the same. Therefore, a general union operation ($R \cup S$) can be defined on pure partition relations without restrictions. However, if cumulative attributes present, a general union needs external information on the common tuples and must be considered as a data manipulation operation that creates a new partition relation with extra input rather than a query operation.

aggregated on the common tuples. Its partition semantics over $\mathcal{U}$ is defined as $\Phi \cup \Psi = (\mathcal{H} \cup \mathcal{I}; \chi'')$ where χ'' is the common extension of χ and χ': $\chi''|_{\pi_B(R)} = \chi$ and $\chi''|_{\pi_B(S)} = \chi'$.

A common use of renaming is together with disjoint union. In this case, compatibility of semantics is ensured automatically:

Definition 16. Disjoint partition union with renaming:
If $X = Z = \emptyset$, the constraint $\mathcal{H}_R \sqcup \mathcal{H}_S$ holds and D does not occur in the database schema, then the disjoint union is a valid operation with renaming*: $\rho_{A \to D}(R) \uplus \rho_{C \to D}(S)$. It is interpreted in the straightforward way. Note that renaming of partition attributes is not defined as a separate operation, it is only valid in this form.*

Definition 17. Common sets union:
If $A = C = X = Z = \emptyset$ and the constraint $\mathcal{H}_R \sqcap \mathcal{H}_S$ holds, the common sets union *$R \Cup S$ of partition relations is a valid operation defined as the traditional union $\pi_B(R) \cup \pi B(S)$ extended with the cumulative attributes Y kept from either relation (not aggregated, they are equal on the common tuples due to the base set constraint). Its partition semantics is defined as of the disjoint union.*

Note that the relations themselves need not to be disjoint in the case of disjoint union but the same symbols must denote disjoint parts of sets (see Figure 4.2, lower right part).

As we can see, validity of some operations depend on functional or base set constraints. Surveying dependency-preservation of operations is needed for building a valid complex expression of this algebra. Although it may be trivial in many cases, a detailed discussion on this topic will be necessary.

4.4.4 Navigation Operations Translated into Partition Algebra

If a data cube is represented by a normalized snowflake schema, operation drill-down corresponds to a right-to-left join and possibly a projection (to omit attributes of the coarser granularity). Roll-up corresponds to a projection (aggregation), possibly combined with a left-to-right-join (to get the attributes of the coarser granularity) and slice corresponds to selection.

Example 9. *Assume the time dimension in Example 7 is decomposed on the relational level along the hierarchy described by functional dependencies $Minute \to Hour$, $Hour \to Day$, $Day \to Month$, etc. A relation J_B corresponds to the basic cuboid (the most detailed data) with the only time-related attribute $Minute$. Another relation J_{HM} holds the $(Hour, Minute)$ pairs, like a dimension table of a star schema. A roll-up from $Minute$ to $Hour$ corresponds to $\pi_{Hour}(J_B \overrightarrow{\bowtie} J_{HM})$ in this case.*

Chapter 5

Structure of Partitions

Section 4.3 introduced the concept of functional dependency into the partition model and showed its importance in the refinement structure. In this section, we a formalism for describing the attribute structure is introduced, based on functional dependencies.

First a structural graph of a single partition relation is considered, followed by the case when a single graph corresponds to two or more relations. The graph notation is then extended to handle different base sets simultaneously and missing ways of partitioning. Schema decomposition and possible user navigation can be driven by the structural graph.

5.1 Structural Graph of Partitions

Design of a partition database on the logical level can be based on the attribute structure. The user can perform navigation operations using attribute names and the structural graph provides the possible navigation paths. Actual relation names are hidden, as graph operations are translated into partition algebra in the background. The graph notation introduced in this section aims to generalize the conventional cuboid lattice graph, in order to provide a theoretical framework for navigation paths in a possibly heterogenous partition database. It can also be used for schema design, but it becomes rather complex for large schemata. Proper adaptations (extensions) of graphical notations like Subject [19] need to be developed in order to support schema design.

A classical dimensional hierarchy can be described by a chain of functional dependencies. This notion is generalized by allowing any closed set of functional dependencies. Moreover, local properties can be expressed by dependencies that are valid under a condition (similarly to [13] but exact selection conditions are used here for specifying validity). Partially applicable attributes and attribute combinations can also be represented by this extended notation.

The possible partitions can be obtained by projection of the multiway partition in the finest granularity. Several projections may, however correspond to the same partition. If, for instance, $A \to B$ holds then partition according to A describes essentially the same partition as AB since B divides no sets further. The only difference is in the naming of sets but the sets do not change. Given the set of functional dependencies that describe the partition structure, the closed attribute sets correspond these equivalence classes of partitions (see Section 4.3).

A set of closed attribute sets w.r.t. a set of functional dependencies is closed under intersection so it forms an intersection-semilattice (meet-semilattice, SL, see Section 2.3.4). It can be represented by a graph according to the partial order of set containment:

Definition 18. Structural graph of partitionings:
Given a partition schema $\mathcal{R}$ and a set of functional dependencies $\mathcal{F}$, the structural graph *of the different partitionings obtained by projection contains the closed attribute sets (w.r.t. $\mathcal{F}$) as nodes and the (nontransitive) instances of set containment as edges.*

A node of a structural graph corresponds to an equivalence class of partitionings (see Section 4.3) and is usually labeled by the closed attribute set it represents (it is unique for each class). However, since the attributes are 'inherited' along the edges, it is enough to indicate the new attributes at each node only. For the sake of clarity all the attributes will be indicated on figures, with the inherited ones in brackets.

A projection of a partition relation corresponds to walking through its graph towards the (closure of the) empty set until we reach the first (smallest) closed set that contains the attributes we project on.

From a multidimensional point of view, the structural graph is the extension of the lattice of cuboids by different resolutions of dimensions of a data cube. The structural graph of a simple hierarchical chain or an alternative hierarchy is the same as the natural way of notation if all the attributes contribute to the hierarchy. Independent attributes result in a more complex but well-structured graph containing all possible ways of partitioning one can get.

For the sake of simplicity, assume each of the partitions have the same cumulative attributes. If this is not the case, the structural graph notation may be extended to indicate the applicability of cumulative attributes as well: a cumulative attribute is attached to a node and is inherited along the paths towards the minimal element (the closure of the empty set).

The following example presents a variety of sets of functional dependencies for a 3-way partition schema. Refer [33] for all possible ternary cases of functional dependencies.

Example 10. *Consider the Post $= (S, Z, D)$ schema of Example 6. Depending on the district and ZIP zone division policy of the city, we get different sets of functional dependencies and structural graphs (see Figure 5.1). The absence of an attribute set in a graph does not mean we do not know the partitioning it determines, but it is equivalent to that of the smallest closed set containing it.*

1. No (non-trivial) functional dependencies *hold, i.e. $\mathcal{F} = \emptyset$. Each set is closed and the structural graph has a complete lattice form.*

2. A simple hierarchy *is given as a zone is a part of a district: $\mathcal{F} = \{Z \rightarrow D\}$. Z appears under D, always together with D.*

3. A simple hierarchical chain *is defined: $\mathcal{F} = \{S \rightarrow Z, Z \rightarrow D\}$. We get the well-known representation of a simple hierarchy if we consider only the attributes outside the brackets.*

4. An alternative hierarchy *is defined: $\mathcal{F} = \{S \rightarrow D, Z \rightarrow D\}$.*

5. An equivalence of attributes *is defined, each zone is equal to a district: $\mathcal{F} = \{Z \rightarrow D, D \rightarrow Z\}$.*

6. A more complex relationship *is defined: each zone is a part of a district, street segments may be shared between districts but not between zones of the same district. The latter condition is formulated by a functional dependency stating that street and district determines zone: $\mathcal{F} = \{Z \rightarrow D, SD \rightarrow Z\}$.*

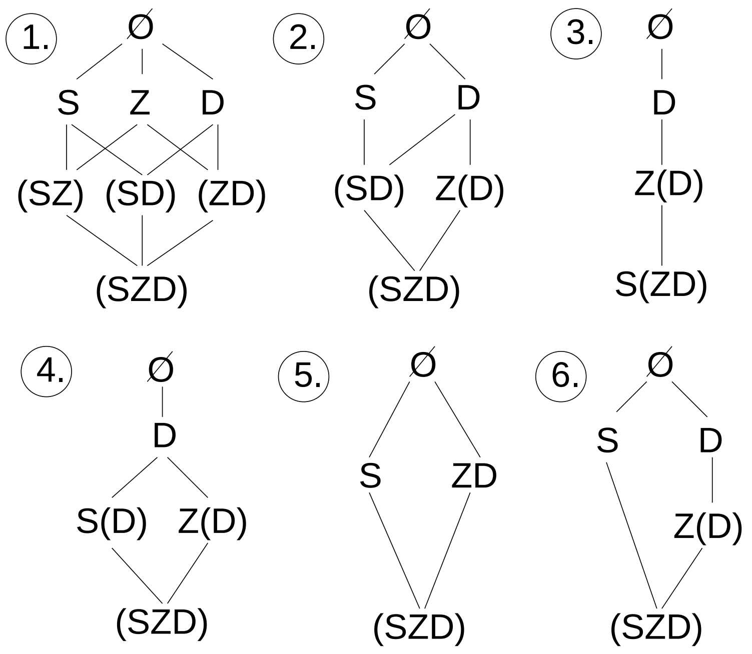


Figure 5.1: Structural graphs for the simplified *Post* schema with different sets of functional dependencies. Nodes represent the essentially different partitionings (see Example 10)

5.2 Extended Structural Graph Notation

5.2.1 Partial Graphs

If two or more relations partition the same set without proper functional dependencies, the full partitioning may not be constructed (see also the join problem in Section 4.2). In this case, a *partial structural graph* can be used to represent the available part of the structure. The partial graph contains the nodes 'known' (i.e. the appropriate partitioning can be constructed). All nodes that are labeled by subsets of a known node are also known (by projection). The known sets are therefore closed under intersection. If a partitioning according to a key is given, each node is known. Otherwise, the structural graph must be cut somewhere horizontally and the resulting partial graph will have several maximal elements. The known maximal closed attribute sets are exactly the closures of the attribute sets of the finest given partition schemata. The following example illustrates a simple case:

Example 11. *Suppose we have restricted information on the partitioning of Example 10 (case #1), knowing which zones and street segments intersect as well as which zones and districts intersect, represented by a partition database with two partition relations $R : (S, Z)$ and $Q : (Z, D)$ (the base set constraint $\mathcal{H}_R = \mathcal{H}_Q$ must be declared). Since neither the 3-way partitioning nor the partitioning according to (S, D) is known, the structural graph is cut at certain points as shown on Figure 5.2.*[1]

[1]However, if $S \rightarrow Z$ and $Z \rightarrow D$ holds then the structural graph is reshaped as a hierarchical chain and the full partitioning can be constructed by join.

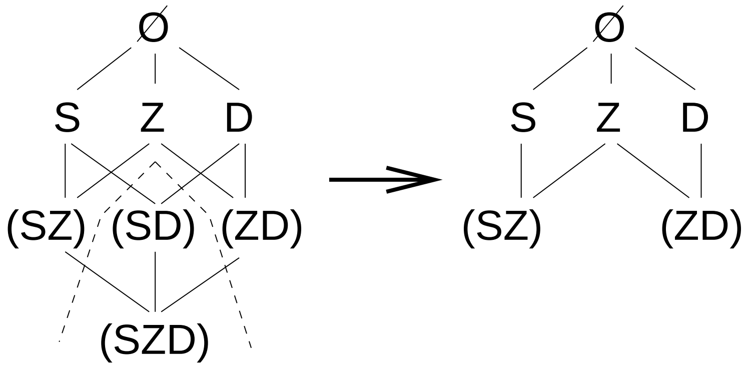


Figure 5.2: Obtaining the partial structural graph of a partition database schema with missing information. The cut indicated by the dashed line is determined by the information available (see example 11)

5.2.2 Subsetting and Equivalence Edges

Partially applicable attributes and local dependencies result in a situation where the global structural graph needs to be refined according to local attributes and dependencies. These local aspects can be included into the global graph by special types of edges.

Subsetting edges (dashed) are used in the structural graph to denote partially applicable attributes or attribute combinations. They represent that a specific way of partitioning (cuboid) is available only for a specific subset. A label indicates the selection condition (e.g. due to a strict base subset constraint w.r.t. a condition θ). If the condition is not specified on the schema level or is not expressible with the attributes of the parent node, the label may be empty. A subsetting edge applies not just to the next node but to all subsequent nodes. A condition of a path is the conjunction of conditions along the path. The graph is consistent w.r.t. the subsetting edges if and only if each node has a well-defined condition, i.e. each path leading from the closure of the empty set to a particular node has the same condition.

Example 12. *Recall Example 7. The granularities $Year(Y), Month(M), Day(D)$ of dimension $Time$ is available for the whole dataset. A finer granularities $Hour(H)$ and $Minute(Mi)$ are only available for this year's data ($Y = 2006$). This can be denoted by a conditional edge as depicted on the left graph of Figure 5.3 (only the structure of $Time$ attributes is drawn).*

Equivalence edges are used to denote local functional dependencies. If a local functional dependency holds, some nodes would be unified in the graph if it were drawn only locally. In fact, they remain separated in the global graph. Such cases are indicated by equivalence edges (double edges) with a condition in square brackets. An equivalence edge is inherited along the graph towards attribute supersets (e.g. if X is equivalent with XY under condition θ, XZ is also equivalent with XYZ under the same or weaker condition). The graph is consistent w.r.t. equivalence edges if and only if each equivalence edge is properly inherited along the global graph.

Example 13. *Assume the ZIP code zones form a refinement of districts in some cites. In others, they have no such hierarchical connection. The global graph becomes similar to the first case in Example 10 but for a well-defined condition the graph looks as the second case. Assuming a selection condition θ specifies those cities, the two graphs can be merged, with equivalence edges indicating this local property as shown in the*

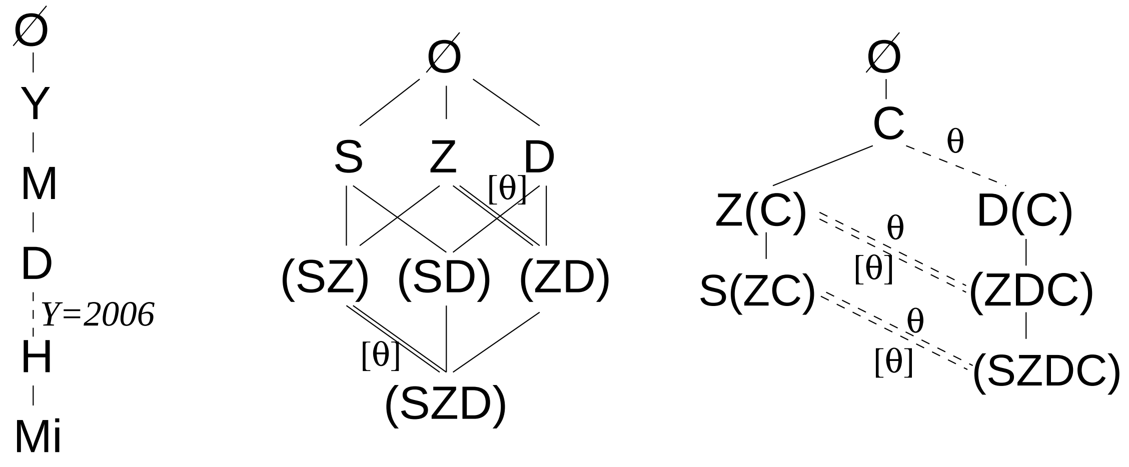


Figure 5.3: Structural graphs with subsetting and equivalence edges (see Examples 12, 13 and 14, respectively)

middle graph on Figure 5.3.[2]

A combination of the two above cases is when districts are not available in each city, but where available, they form a coarser partitioning than zones of ZIP codes. This is illustrated next.

If a functional dependency exists so that an attribute A with base set $\mathcal{H}$ is dependent on an attribute B with a larger base set $\mathcal{I}$, the dependency $B \rightarrow A$ is not applicable in $\mathcal{I}$, only in $\mathcal{H}$. Therefore, the closure of B will not contain A in $\mathcal{I}$, just in $\mathcal{H}$. But the two closures form an equivalent partitioning over $\mathcal{H}$. To indicate this, *subsetting equivalence edges* (double dashed) are used in the structural graph:

Example 14. *Recall Example 8 of sub-partitioning and assume a hierarchy $\{S \rightarrow Z, Z \rightarrow C\}$ holds for the whole country and additionally, a consistent hierarchy $\{Z \rightarrow D, D \rightarrow C\}$ holds for big cities. Assume that a selection condition θ on cities selects the big ones. The structural graph is shown on the right side of Figure 5.3 with a conditional edge for big cities. ZC and ZDC are equivalent partitionings over the big cities but they cannot be represented as a single node since D is not applicable to non-big cities. The equivalence is indicated by a special edge. The same holds for SZC and $SZDC$.*

A subsetting equivalence edge has a subsetting condition and a (possibly stronger) equivalence condition. In most cases they do not differ and so the equivalence condition can be omitted.

5.3 Schema Decomposition

A partition relation schema can be decomposed along functional dependencies over partition attributes as a traditional relation, in a cumulation-preserving way. To ensure a decomposition (e.g. BCNF or 3NF, see [2]) of a partition schema is semantically correct, base set equality constraints (see Section 4.3) must be declared. The structural graph is assigned to the decomposed partition database schema instead of a single relation schema and indicates the ways of partitionings available by projection and join. [71] and

[2]If we had partial information as in Example 11 the full information would be available by left-to-right join for the part of the dataset determined by θ.

[70] consider derivation of partitionings by functional dependencies in a deductional formalism. Here an algebraic-operational approach is used (refer to the operations in Section 4.4).

Example 15. *A valid decomposition of the schema of the second case in Example 10 is* $R : (S, Z)$*;* $Q : (Z, D)$ *and the nodes of the graph can be generated using algebra operations as follows:* $\emptyset : \pi_{\emptyset}(R) = \pi_{\emptyset}(Q)$*,* $S : \pi_S(R)$*,* $D : \pi_D(Q)$*,* $(SD) : \pi_{SD}(R \overset{\rightarrow}{\bowtie} Q)$*,* $Z(D) : Q$*,* $(SZD) : R \overset{\rightarrow}{\bowtie} Q$*.*

In some cases, a redundant decomposition may be convenient to speed up some operations (like materialized views). In such a case, an equal base set dependency is needed again. The schema designer should check whether the partitioning for each node of the structural graph can be generated using algebra operations, and which way to prefer if there are multiple choices. The generating expressions are then associated to the nodes.

The following procedure is advised as a semi-automatic process of decomposition, given a structural graph (with the extended notation), to create the relational partition schema. The process lets the designer decide at some steps whether the schema should remain more or less redundant to reduce computation complexity for frequent or high-cost queries (e.g. some joins). It is important to add proper base set constraints for each new partition schema introduced.

1. **Determining the critical nodes.** Each maximal node of the graph and each node that has only subsetting edges downwards is *critical*. If they are known (the corresponding partitioning [cuboid] can be computed) then all other nodes are known, by projection (aggregation). Redundancy, efficiency is not considered in this step yet.

2. **Assigning relation schemata.** Assign a matching partition relational schema to each critical node and add proper base subset (or equality) constraints.

3. **Decomposition along attribute cover.** This step corresponds to a possible local normalization. Attributes are always inherited along a subsetting edge, therefore, they 'cover' the subset. The neighboring critical nodes are compared whether one can be decomposed as a left-to-right join with the other (some attributes can be eliminated from the subset schema). The designer can decide whether such a decomposition is reasonable and does not lead to inefficiency.

4. **Decomposition along relation cover.** This step can be processed together with the previous one and has an opposite effect. Part of a cuboid can be computed from subset data via aggregation along a subsetting edge. The cuboid with larger base set is then decomposed as the disjoint union of cuboids of its subsets.

5. **Decomposition using non-critical nodes.** Consider non-critical nodes in this step whether it is reasonable to assign own partition schemata to them. It can be a global normalization along functional dependencies (see Section 10.3) up to a phase that does not yield much inefficiency by computing queries, which can be interpreted as separating dimension tables from the fact table (together with the next step). As an opposite direction (denormalization), the designer may decide to introduce relation schemata that are redundant, in order to improve efficiency. Note that these normalization-denormalization issues do not affect the structural graph and the user navigation facilities, only the way (and complexity) of computation of navigation queries and updates.

6. **Eliminating aggregation redundancy.** Up to this point, each partition schema had to contain all cumulative attributes. This step is for eliminating some of them. A classical case is to drop all cumulative attributes from schemata corresponding to non-critical nodes since they can be computed

by aggregation and join. The designer has the freedom here to keep some of them as redundant, again, to improve computation efficiency of frequent queries.

7. **Associating algebra expressions.** Each remaining node gets a partition algebra operation that is used to compute the appropriate cuboid. The critical nodes have their own partition schemata or algebra expressions already as results of previous decompositions (steps 2-4, disjoint union or left-to-right join). The remaining nodes can be computed by projection (aggregation), determined by the graph structure. This step is not necessarily explicit.

5.4 User View of a Structural Graph

5.4.1 Separation and Dimensions

The structural graph in its basic form can be complex as the number of attributes raise and the attributes are not in a hierarchical chain. Two types of separation are considered for simplification: horizontal and vertical separation.

Horizontal separation: Assume the structural graph has a node that is crossed by all possible paths connecting the (closure of the) empty set with the full set. This node corresponds to a critical way of partitioning and the structural graph may be separated at this node. The concept of this separation is introduced in [28] (see also [61]) as *direct product decomposition* of the semilattice of closed sets. The existence of such a node is usually difficult to capture by looking at the functional dependency notation only. If the closed sets are direct product decomposable, the critical set of decomposition is probably the best choice for most cases to be stored as part of a redundant relational decomposition (materialized view).

Vertical separation is more common with multidimensional schemata: distinct attribute groups are identified as *dimensions* and the graph is separated vertically (the separated components should be treated together in parallel, e.g. while querying).

Note that not every graph can be separated.

Separation allows a better view of the structural graph when it becomes complex due to relative independence of attributes. For data modeling and schema design, the process is reversed: one designs the dimensions as separate graphs and they are combined together to achieve the structural graph of the whole schema.

The concept of dimension may be based on such separations as it matches the traditional dimension concept. There can be graphs that can be separated horizontally and the horizontal components can be separated further, vertically. This models *nested dimensions* mentioned in Section 3.2 and relativizes the dimension concept.

5.4.2 Navigation

A simple partition query consists of a selection and a projection (aggregation). The selection condition restricts the base set. Each valid query has a corresponding path in the structural graph that starts from the (closure of the) empty attribute set and leads along the edges to supersets of attributes until the node of the smallest attribute set that contains the attributes of the query (both in the selection and in the projection specification) is reached. If no such node exists, the needed partition is not known and the query is invalid. Moving along a subsetting edge is only valid if the selection condition of the query is at least as strict as of the edge. Equivalence edges are automatically traversed if the selection condition implies the equivalence condition of the edge. After the appropriate node is found, the system can translate the query to partition algebra: the expression associated with the node (decomposition step 7) is used to construct the full partition over a base set and the specified selection and projection is applied.

Starting from a particular node with a simple expression, a roll-up-like operation is performed by omitting some attributes from the current projection (aggregation) that leads to move along an edge in the graph. Drill-down is just the opposite. The subsetting edges determine the validity of drill-down operations: drill-down along them is only possible in a slice of the cube.

Example 16. *Refer to the second case of Example 10. A query to determine which district contains a given zone x can be formulated as $\pi_D\sigma_{Z=x}$ over the main base set of the database. The corresponding path is the one leads directly from the empty set to the node $Z(D)$. Consider the decomposition according to Example 15 where relation Q is attached to node $Z(D)$. This way the query is translated to partition algebra as $\pi_D(\sigma_{Z=x}(Q))$. Under a different decomposition a different algebraic query may be generated but it can be hidden from the user.*

Example 17. *Recall Example 7. The schema was decomposed by partial relation cover (step 4). To get the aggregated data (by days) of journeys during December 2005 and January 2006, one may use the query $\pi_D\sigma_{M=12/2005 \vee M=01/2006}$. It is translated as $\pi_D(\sigma_{M=12/2005}(Journey_{old})) \uplus \pi_D(\sigma_{M=01/2006}(Journey))$. If the selection is restricted to January 2006, one may drill down to hours' granularity ($\pi_H\sigma_{M=01/2006}$, translated as $\pi_H(\sigma_{M=01/2006}(Journey))$). Including December 2005 it is not possible.*

Chapter 6

General Partition Model

This chapter presents a proposal of a general partition database model and illustrates its capabilities with sample cases. It is based on the concepts discussed in the previous chapters. It is aimed to be the basis or starting point for the formal foundation. In order to survey all the questions and points, further work is needed.

6.1 Structure of the Partition Database Model

A partition database models the real world as a system of disjoint sets and stores this set structure together with aggregation data. Foundations of this model was presented in the previous chapters with partition semantics, functional and base set constraints, algebra operations and structural graphs. In order to construct such a database and to allow more complex operations, they must be put together in a compact framework. This chapter sketches such a structure and a set of operations (basic partition language), based on some example cases. The aim is to demonstrate what kind of operations need to be implemented in a partition database system. The language can be improved in the future by considering more sophisticated practical examples. Syntax is not fixed: the language can be extended with *plug-ins* for specific parts, depending on the application domain. The chosen plug-ins affect the expressivity of the language.

The partition database model can be put into a four-level framework. The highest level is the partition structural level, where the conceptual structure of partitioned base sets and their partition attributes are considered. Theoretically, this can be populated with raw data, but it is usually not known in full detail. The next level is the logical administrative level, which defines the available information and assigns partition relation schemata to the partition description. Actual data of partition relations appear on the instance (third) level and partition semantics belong to the interpretation (fourth) level. The first level is in the main focus, and additionally the second, others are not considered deeply.

6.1.1 The Structural Level

A partition database is organized along base sets and their partitions. Each set can be partitioned in several possible ways, an elementary partition corresponds to an attribute. The structural level describes the partition attributes of the sets as well as the aggregation attributes. These form the possible partition combinations (overlay partitions). Attribute values taken from a *domain of symbols* give the set names of partitions, but the actual data belongs to the instance level.

In the generic model, no structure of symbols is assumed. However, it it a possible extension of the model to assume a structure of the set symbols and use its facilities in the partition language (*set symbol language*). For instance, if the symbols assumed to be strings, concatenation operation can be a valid relabel function for merging attributes. Another structuring paradigm is to define elementary set names and compound set names in order to reflect semantics (eg. how a set is constructed). A nice example for both is to define a concatenation operation with a separator symbol '$\cap$', reflecting the intersection construction. For the sake of simplicity, the generic model is taken, no structure is assumed for the domain of attribute symbols and each name is elementary.

Theoretically, finest-granularity raw data can be assigned to the partition structure but this is not always known. Instead, the structural level contains a conceptual structure: which attribute is applicable to which set. The logical level will describe which partitions are available (*known partitions*). They form a substructure of the conceptual model on the structural level.

There is a formal superset of all the sets modeled, called the *universal base set*. The structure is recursive, that means, any subset can be considered as a base set and partitioned further by chosing a subset symbol from an arbitrary partition. Each set automatically inherits the partitioning of its supersets and may extend it with extra attributes and extra dependencies on the new attributes. The available sets can be reached from the universal base set using a *set definition expression*. The format of set definition expressions is another possible 'plug-in' of the model. Simple set paths are considered now but more complicated constructions can be studied in the future. The key is to indentify a set without ambiguity. Moreover, a referred set must be constructed out of the elementary set intersections and must rely on known disjointness relationships.

A *set path expression* used now for set definition is a nested set sequence, starting from the universal base set. At each level a single partition attribute is chosen and a subset symbol of it. Thus, set path expressions form a tree structure. Some expressions are equivalent, yielding a lattice (DAG, directed acyclic graph) structure of base sets. The nodes correspond to different base sets and are labeled with their own partition (and aggregation) attributes, while each edge is labeled with the partition attribute of the parent node and a set symbol of its domain that defines the child set. If the equivalence of two set definitions cannot be verified using the known information, they are considered as defining different sets.

Refinement relationships among partitions are also considered on this level (as functional dependencies). Constraints are usually global in the scope of an attribute, but a constraint is always assigned to a base set. This allows local dependencies and horizontal decomposition [13]. The impact of local dependencies is not studied in detail yet but it is a promising direction since the aim is to take full advantage of partial information. With a local dependency, some partitions not fully available may be computed for a specific subset at least.

Chapter 5 introduced the *structural graph* notation where each node corresponds a possible aggregation and the graph specifies the aggregation paths. There are two possible ways to use the structural graph notation. The first one is to use the extended notation with subsetting edges. This way a global graph can be constructed. A subsetting or equivalence edge is either labeled by an atomic equality condition or is not labeled (not expressible by the available attributes of the node). If no local dependencies allowed, equivalence edges become invalid. Subsetting equivalence edges may, however, still be necessary to express dependencies about attributes defined on a subset only.

The other way is not to use subsetting edges but to have different full graphs for base sets. If a subset has its own partition attributes then the subset has its own graph, which must be consistent with the graph of its superset. A structural graph on this level is always a full graph, i.e. there is only one partitioning with the finest granularity for each base set because since the available partitions (known data) is not considered yet, just the possible structure and aggregation.

It is a question how to represent the structural graph – either the first or the second or another way. For the sake of simplicity, signatures of partition operations will be introduced in the second way but the actual representation of the structure may differ.

Some attributes can be defined as *derived attributes*. Syntax of derivation formulae is part of another possible plug-in, *partition definition expression*. The chosen syntax for the current model will be discussed in Section 6.2.2.

The notation is extended with *aggregation attributes*. Only cumulative attributes with default cumulation operators are considered yet (see Section 4.1): a cumulative attribute can be attached to a node of a structural graph and is inherited along aggregation paths (attribute subsets). Further extensions may include more sophisticated aggregation handling by blocking inheritance along specific paths to meet semi-summarizability 3.3, to define algebraic aggregations as derived attributes, or to attach a holistic attribute to nodes one-by-one. Although not considered yet, *aggregation definition expression* is a possible future plug-in facility, similarly to set and partition definition expressions.

Attribute names are grouped into *namespaces*, which correspond to concepts (eg. a namespace 'time' may contain attributes like 'year', 'week', etc). They may be used as a basis for scope of names (e.g. global, user), access rights and persistency (temporal, permanent). Namespaces allow the grouping of attributes (e.g. dimensions of a multidimensional schema can be modeled by namespaces and functional dependencies). A temporal namespace allows more complex query workflows in sessions: the results are not stored in the database. An attribute name must be unique in a namespace. Global aliases for sets in a namespace can be introduced in order to make references to them easier. Namespace structure is not considered in detail but it is important to note that a temporal namespace must not be referenced by a persistent namespace.

To summarize, this level must contain the following structures:

- A set $\mathcal{S}$ of possible elementary set names. These symbols can be used as values of attributes in the database. If a structure (e.g. algebra) is assumed over them, set name usage (*set symbol language*) is a possible plug-in for the model (including ordering and relabel operations, compound names – and possibly reflecting semantics);
- A base set symbol $u \in \mathcal{S}$ denoting the common superset of all sets represented by the database (universasl base set, the only set symbol in the database outside a partition relation);
- A set $\mathcal{N}$ of namespaces corresponding to concepts;
- A *set definition* and a *partition definition* formalism (as *plug-ins*);
- A partitioning attribute system $\mathcal{P}$:
  - Attributes can take their values from $\mathcal{S}$ (attribute domain),
  - Each attribute name is assigned to a namespace of $\mathcal{N}$ (names must be unique in a namespace),
  - A set definition expression is assigned to each partition attribute, denoting the (primary) base set of the attribute[1] – this must form a proper subset structure (no cycles allowed w.r.t. attributes),
  - Optionally, a partition definition expression can be assigned to one or more partition attributes (these attributes become *derived*) – definitions implied by these expressions must be acyclic,
  - A set $\mathcal{F}$ of functional dependencies over attributes of $\mathcal{P}$ defining the refinement structure of partitions (each constraint is declared with a base set definition expression)[2]

[1]An attribute symbol is applicable to its primary base set or any definable subset of it (inheritance). For the operations, a partition is always considered together with its base set. Partition of the primary base set is the *root ancestor* of an inherited partition with the same attribute name.

[2]Although local dependencies are allowrd, the information held by them is not fully exploited at the current stage of the current model.

- A set $\mathcal{A}$ of aggregation attributes (with aggregation operators), each attribute assigned to a base set and a maximal partition combination to which the aggregation is conceptually meaningful. At the current stage, only cumulative (distributive) aggregations are allowed. This can be extended, including a plug-in for aggregation definition.

This forms a conceptual structure and can be used for navigation and schema construction operations. The three important parts are: the base set structure by attributes (tree of base set definitions, forming a directed acyclic graph of base sets), the attribute derivation structure (can be represented by a directed acyclic graph) and the structural graph(s) (either a global graph with subsetting edges or several graphs for base sets with different attributes). This level declares a conceptual or 'ideal case' structure. The actually available ('known') information is declared on the next level and covers only a subset of this structure if some information is missing (some partitions are unknown).

This abstract framework can be instantiated by choosing a sub-language for each of the indicated plug-ins and constructing a partition language based on them. The basic partition language (Section 6.2) and the current examples are worked out based on the following plug-in choices:

Set symbol language : generic model, no structure of set symbols is assumed by default. In one case (explicitly noted) a string type is assumed with a concatenation operator as a demonstrative example.

Set definition expression: Set path expression (universal base set u or a partition-subset path, see *SetPath* below in Section 6.2.1). Set path expressions form a tree structure (considering equivalent set paths, the base set structure is a DAG).

Partition definition expression: Definition formulae for derived attributes (see Section 6.2.2 below). Attributes of the same base set (or inherited attributes) can be used in a definition, as well as attributes of its subsets defined as base sets by other attributes of the formula.

Aggregation definition expression: Not considered in this simple model, only cumulative attributes exist. Future prospects are sketched in Section 6.5.5 regarding to aggregation handling.

This structural level may also be considered as an extended metadata level, because the structure of partitions express data about subset-level data. It hides actual partition relation names and provides an interface to the database based on the partition structure, in terms of base sets and partition attribute combinations, so that the user does not need to consider how a particular partition can be constructed from partition relations.

6.1.2 The Logical, Administrative Level

The logical level defines the partition schema as in Chapter 4.1, attached to the structure declared in the structural level. An algebra expression must be attached to each node of the structural graph. The decomposition method sketched in Section 5.3 can be used here. The expression for most of the nodes can be obtained automatically as the projection (aggregation) of an underlying critical node, or a left-to-right join expression if the schema is normalized. Validity of attached algebra expressions can be verified by base set constraints that follow from the structural graph.

An atomic relation used in these algebra expressions can be of type external, derived or in-line. External means the actual data is not defined on this level (it is a relation schema), but on the instance level. Derived relations are based on the attribute derivations defined on the structural level and can be computed on-the-fly (at least theoretically) and in-line relations are constant relations defined on this level. Some of the derived attributes may not be part of any derived relation schema. The attribute definition becomes an integrity constraint in such a case.

Please note that nodes of this semilattice graph correspond to the equivalence classes of partitions (see Section 4.3) and therefore, some attribute combinations do not have their own node but they are equivalent with the node that represents their closure.

There may be nodes of the structural graph without labels of expression, i.e. there may be partitions that are not computable from the known information. In such a case, the actual structural graph becomes partial as in Section 5.2.1, containing the available partition combinations, where some sets may have several maximal nodes (partitions of finest granularity). As soon as the information corresponding to the unlabeled node is available, the node can be labeled and the partition becomes 'known' (see also Section 6.2.3 below). As new data is loaded, parts and structures defined by set definition and partition definition expressions are automatically created, based on the new data.

Based on the structural level, this level contains the following elements:

- A set $\mathcal{R}$ of partition relation symbols with schema specification (partition and aggregation attributes), each of them typed as either external, derived or in-line.
- Assignment of a valid partition algebra expression to each derived relation symbol as a definition of the derived partition relation. They must match the schema defined in the structural level (the attribute derivation and refinement specifications).
- Assignment of a constant partition relation to each of the in-line relation symbols.
- Assignment of a partition relational algebra expression to the nodes of the structural graph specified on the structural level (see also Section 5.3 and 5.4.2), in a consistent way.[3] This assignment is not necessarily total, the unassigned nodes remain unknown.

The main point of this level is that is describes the “known” part of the structural graph taken from the structural level (see partial structural graphs in Section 5.2.1) and specifies the partition relational schema that must be populated with data on the instance level. If a user asks a query for a specific way of partitioning, the decision is made on this level whether the result is unknown or can be computed using the data of the instance level, and how it can be computed or retrieved. The partition relation schema may be reorganized (e.g. normalized or denormalized) without affecting the user operations on the structural level, where the relation names are hidden from the user. Relation names can be handled on the logical administrative level, since the structural level contains a conceptual description and should remain independent of the actual relational decomposition.

6.1.3 The Instance Level

The instance level assigns data to partition relational symbols specified on the logical level by resolving the symbols as data sources.

Consistency of data must be validated against the structural graph, which can be used for the basis to specify integrity constraints. Base set constraints (see Section 4.3) and functional dependencies are examples, but completeness of constraint formalism against the partition structure needs further work. Refinement consistency (functional dependencies), set consistency (consistency of the same partition described by from

[3]In order to meet disjointness conditions implied by the structural level, the constant empty partition relation may be assigned to some of the nodes.

different relations, e.g. redundant decompositions or inclusion dependencies for repartitioning subsets) and aggregation consistency are the three major areas.

This level may also allow specification of how the partition data is obtained from different sources (e.g. from OLTP databases, ETL) or some partition relations to be declared as 'stubs' so that their data can be loaded or computed on-demand when needed. The model focuses on relational representation because the basic idea behind partition relations is to simplify the naming of sets (a tuple represents a set intersection and can be used as a name of the set). In principle, multidimensional representation may also be used. Different data representations of the underlying population may also be an issue of this (and also the interpretation) level. For instance, if a space is modeled as an infinite set of points and the spatial data representation is vector-based or constraint-based, computation of a partition relation should be specified as a spatial overlay operation.

6.1.4 The Interpretation Level

The interpretation level is the level of partition semantics (see Section 4.1). The instance level specifies the data and how it is obtained, while here the point is what the data refers to. Symbols of partition relations must be resolved in terms of the the population-level data (set names assigned to real sets and/or the semantics of partition attributes as metadata).

6.2 Basic Partition Language

Based on sample cases, some partition database operations will be defined. The list of proposed operations can be found in Appendix A. Future work is needed in order to determine the expressive power or to be able to state completeness in any sense. The syntax and expressivity of the language is based on the choice of plug-ins (see the plug-in options in Section 6.1.1). The aim is to demonstrate the kind of types and operations for the partition database model.

6.2.1 Types and Type Construction

This section introduces the notations of basic types and type constructors used by partition operations. No set symbol language is used (generic domain) and the set definition formalism is fixed as the simple type $SetPath$ below.

- *SetSymb* (set symbol): e.g. $a, b, ...u, ...$ (usually denoted by lowercase letters): Domain elements of partition attributes (members of $\mathcal{S}$, see Section 6.1.1). They denote subsets and base sets. These symbols are treated as local w.r.t. a partition attribute.
- *AttrSymb, PartSymb, AggregSymb* (attribute symbols, either partition or aggregation attributes, resp.): e.g. $A, B, ..., X, ...$ (usually denoted by uppercase letters): Attribute symbols of partition relations (either partition or aggregation attributes). Each attribute symbol always belongs to a namespace.
- *AggregCumulOp* (Cumulative attribute with operation), e.g.. $(X, +)$: aggregation attribute with default cumulative operation. The operation is used to compute the value for a disjoint union based on the values of unified sets. Only this type of aggregation is supported in the current stage.
- *Partition* (Partition[combination], corresponds to a partition relational schema), e.g. $[A, B, C; X, Y]$: multiway partition by several partition attributes (optionally extended with aggregation attributes, separated from partition attributes by a semicolon). The fully aggregated partition

(the partition by the empty set of attributes) is denoted by [] (with cumulative attribute X it becomes $[;X]$).

- *PartSet* (set symbol + partition, a partitioned set), e.g. $h[A,B,C]$, A partitioned set: a set symbol extended with partition notation.
- *SetPath, PartPath* (set definition, partition specification): a reference to a particular set by its definition path, i.e. how the set is reached from the global superset through selecting subsets from partitions.[4] An example for a set is $h[A].a[B].b$. A set definition can be extended with a partition specification, eg. $h[A].a[B].b[C,D;X]$[5]. The constant empty set or any of its partitions is denoted by the constant $\emptyset$. In principle, this set definition formalism may be extended or substituted by another, more expressive syntax.
- *FunctDep* (functional dependency): e.g. $AB \rightarrow C$, declares partition A, B as a refinement of C in a partition structure. Only global dependencies are considered yet. Therefore, the base set of a constraint is always the intersection of the base sets of its attributes.
- *PartStruct* (partition structure, semilattice): structure of the partition attributes for a specific base set as closed attribute sets w.r.t. functional dependencies. It contains the different partitionings (attribute combinations) as a structural graph with global cumulative attributes. The structure is not partial, i.e. it contains a maximal closed set with all the attributes (corresponding to the finest granularity). It describes a structure which is valid for a whole set. Subsetting or equivalence edges are not used but the type may be extended if necessary.
- *HyPartStruct* (cut partition structure): an extension of PartStruct: it can be partial, corresponding to the 'known' partitions obtained in the logical level (it can express incomparable partitions). It can contain several maximal elements. Cumulative attributes may be attached to any of the nodes and are inherited along paths towards the minimal element (closure of the empty set). A PartStruct value can also be treated as HyPartStruct.
- *PartDataset* (partition relation): e.g. *Rel*, partition relation with partition schema, possibly with aggregation attributes. The UNKNOWN special constant is of this type, denoting an unknown partition by an attribute combination.
- *ExternDatasource* (external data source): e.g. *ExtRel*, a symbolic data source specification for the instance level, 'producing' a partition data set. While a partition dataset contains the data itself, an external data source refers how the data is obtained (e.g. a table or view name in an external database, or a raw data file).
- *Logical*: Boolean value, it can be YES, NO or UNKNOWN.
- *Condition*: A subset selection condition over a base set. Atomic conditions are attribute equality and inequality with another attribute or a constant value. Compound conditions can be formed with logical connectives $(\wedge, \vee, \neg)$.
- *PartAlgExpr*: A partition algebra expression (see Section 4.4).

[4]As a syntactic sugar, global aliases may also be defined in a namespace to make some references simpler.

[5]This partition specification is a partition schema declaration. Please note this is not the same as a partition definition expression, which is used for explicit definition of a partition (by derivation)

- *PartDatabase* (partition database), e.g. $\mathcal{D}$: a reference to the actual partition database as an abstract data type. Most of the operations work on a single database, which will be taken as an implicit parameter of them to simplify the notation.
- Set construction: each of the types above can be used in set type construction, i.e. set of set symbols, attribute symbols, constraints, etc. Set construction is denoted by $\{\}$ for both type and value specification, e.g. $\{SetSymb\}$ as a type construction or $\{a, b, c\}$ as a value of this type.
- Tuple (direct product) construction: a tuple type can be declared, denoted by $()$. For instance, *AggregCumulOp* can be interpreted as a tuple construction of an attribute symbol and an aggregation operation.

6.2.2 Partition Attribute Definition

It can be treated as a special type but it deserves a separate subsection. A simple definition sub-language is introduced as a plug-in for the abstract partition framework.

A definition on the structural level is valid for a single attribute A and the type of such an attribute definition is formally denoted by $AttrDef(A)$. Definition of a derived attribute is an assignment, e.g.: $A = \alpha$. The right-hand side α can be constructed of:

- Reference to a constant $SetSymb$ value.[6]
- Reference of another attribute that is valid over the base set (*PartPath, PartSymb*).
- Using an n-ary function f over the symbols (relabeling function).
- A case-switch statement, e.g. $case(A = a : B = b; A = a' : B = b'; else\, B = b'')$. The conditions are of type *Condition* and must define mutually disjoint subsets (a partition). Three different semantics can be introduced: 1. if disjointness of conditions is not implied by the partition structure then the whole definition is invalid, 2. if disjointness is not implied by the actual data then the definition is invalid (in this case a data manipulation may invalidate the definition and validation facilities must be provided), 3. if two overlapping subsets exist then the definition is ambiguous for that part and gets $\top$ as a default value. One of these semantics must be chosen in a particular context.

  The *else* special condition may be used with natural semantics. If the conditions does not cover the whole set, a default branch $else : B = \bot$ can be attached.[7]

Remarks:

A definition is based on already defined or non-derived attributes. An attribute may not be redefined or overloaded. This ensures the acyclicity of the attribute derivation.

The formalism can be extended. For instance, if a set symbol structure and language is assumed, the partition definition formalism can exploit its facilities (eg. ordering or other predicates, operations).

Aggregation attribute definition is a future issue. It may be performed similarly, but more sophisticated conditions and operations are needed (e.g. aggregation type, interpretation of derived aggregation). To

[6]Special constants may be used when needed as proposed, as part of a set symbol language plug-in: $\bot$ [unknown, default, out-of-domain or not applicable values] and $\top$ [insufficiently specified or ambiguous value, e.g. if an assumed disjointness relationship does not hold], in order to prepare the definition for unexpected data manipulations (if definition of a derived attribute is based on an occasional property of data, which is not specified or constrained by the structural level).

[7]Another extra keyword *more* may also be introduced, referring to the overlapping cases if the conditions are not disjoint.

achieve this, the aggregation attribute concept must be extended beyond cumulative attributes and the aggregation attributes must be connected to the population by a symbolic aggregation formula, similarly to partition semantics for partition attributes. Interpretation of aggregation can be based on such formulae. On the structural level, aggregation definition formalism can be included for the definition of algebraic aggregations. This extension is needed to reach completeness in expressing multidimensional schemata.

6.2.3 Valid and Known Partitions

A partition given by its base set and attributes is *valid* if each of the attributes is declared (or inherited) for that base set on the structural level. If it does not hold, it results a syntax error. A more strict condition is whether the given partitioning is *known*, i.e. available, computable.

For instance, if partitions $h[A, B]$ and $h[B, C]$ are available as partition relations then $h[B]$ and $h[A, B]$ are valid and known. Partition $h[A, B, C]$ is valid too, but it is known only if it can be computed (e.g. if the constraint $B \to C$ holds, i.e. B refines C). $h[D]$ will not be valid unless partition D is explicitly declared for the base set h. What is known depends on the logical (administrative) level, whether the corresponding node of the structural graph is labeled with a partition algebra expression.

Attribute combinations of valid but unknown partitions allow external information being added to the database. Consistency must be checked against the known information. Ensuring aggregation consistency can be a costly operation but necessary for database integrity. For instance, a traditional relational join operation of existing partition relations gives an upper bound for the external information [71]. In the above example, $h[A, B, C]$ must be contained by $h[A, B] \bowtie h[B, C]$.

If a partition $h[A, B, C]$ is referenced by a query, it is translated to the partition algebra operation attached to the appropriate node on the logical level. Some examples are given in Section 5.4.2.

6.3 Sample Cases and Operations

The following sample cases illustrate the capabilities of this framework and some future prospects. Some operations are defined based on them. A list of possible operations can be found in Appendix A. The signatures follow the types introduced in Section 6.2.1 and a parameter referring to the actual database is assumed implicitly. Semantics are described informally. The complete specification and implementation of the basic partition language is a future issue.

6.3.1 Simple construction

Assume a base set h is given with three partition attributes A, B, C and a cumulative attribute X with operation '+'. Partitions form two 'dimensions' since B is a refinement of C: $B \to C$. The partition database is created and the data is imported. Two cases are considered regarding the data source:

1. The 3-way partition $(A, B, C; X)$ is stored in an external table *Rel*:

   InitSchema(h)
   DeclarePartition(h,PartStructCreate($[A, B, C; X], \{B \to C\}, \{(X, +)\}$))
   ImportPartition($h[A, B, C; X], Rel$)

2. Table Rel' contains data of $(A, B; X)$ and data of partition C is read from table Rel'' with schema (B, C), due to the refinement relationship $B \to C$:

InitSchema(h)
DeclarePartition(h,PartStructCreate,($[A, B, C; X]$, $\{B \rightarrow C\}$, $\{(X, +)\}$))
ImportPartition($h[A, B; X]$, Rel')
ImportPartition($h[B, C]$, Rel'')

DeclarePartition operates on the structural level, while ImportPartition operates on the logical level. ImportPartition must check for the validity of $B \rightarrow C$ in Rel''. After a successful import, $h[A, B, C; X]$ becomes a valid, known partitioning. It can be treated as a view defined by the partition algebra expression $Rel' \overrightarrow{\bowtie} Rel''$, which is attached to the corresponding node in the structural graph on the logical level.

6.3.2 Querying a partition relation

The basic query is to retrieve the partition relation for a specified base set and attributes. It corresponds to a 'SELECT-FROM' query but a base set specification is given instead of a table name:

Rel =QueryPartition($h[A, C; X]$) – retrieves the appropriate partition relation according to the attached algebra expression on the logical level if it is known. The result may be the special constant UNKNOWN if the specified attributes are valid but the partition by their combination is not known.

6.3.3 Querying the partition structure. Navigation

To query which attributes are valid for a base set and which of their combinations correspond to a known partitioning, schema query operations can be used. Some examples follow.

- Valid partition schema and refinement relationships for base set h (partition structure):

  $\mathcal{G}$ =GetPartStruct(h) – retrieves the valid (declared) partition structure of h (partition attributes, functional dependencies and aggregates declared for the full set h on the structural level). The result in this case is: $\mathcal{G} = ([A, B, C; X], \{B \rightarrow C\}, \{(X, +)\})$.

- Query the schema of the finest known partitions of h:

  χ =GetKnownPartitions(h) – retrieves a set of partition schemata that are 'known' for h (according to the logical level). The finest partition is known for h in the current example, the result is therefore a single schema: $\chi = \{[A, B, C; X]\}$

- The above operation can have an extended syntax for specifying a partition schema. It gives the known partition schemata of h that are extensions of the given schema:

  χ =GetKnownPartitions($h[A; X]$) – retrieves the schemata of the finest known partitions for h containing A and X.

- Querying refinements (drill-downs) and aggregations (roll-ups) for a specified partition schema:

  χ =GetRollups($h[A, C; X]$) – retrieves: $\{[A; X], [C; X]\}$ (one-step aggregations)
  χ =GetDirectRefinements($h[A, C; X]$) – retrieves: $\{[A, B, C; X]\}$ (one-step refinements)
  χ =GetDrilldowns($h[A, C; X]$) – retrieves $\{[A, B; X]\}$ (one-step refinements without the redundant [refined] partitions)

The navigation operations of a traditional data cube can be achieved using these operations and QueryPartition. Dimensions implicitly follow from the overlay partitions. If two attributes are functionally independent then they belong to different dimensions. If there is a need for explicit dimension handling beyond this, namespaces must be used and extended syntax of the operations may be needed (e.g. GetRollUp restricted to a namespace as dimension specification).

6.3.4 Incomparable partitions. Resolving by external knowledge

Assume a base set h is given with partition attributes A, B, C without any refinement relationships among them and an additive cumulative attribute $(X; +)$. However $h[A, B; X]$ and $h[B, C]$ are the known partitions of h and $[A, B, C]$ cannot be computed.[8]

Example operations:

- Query the known partitions for h:

  χ =GetKnownPartitions(h) – retrieves the finest known partition schemata of h: $\{[A, B; X], [B, C]\}$.

- QueryPartition for $h[A, B, C]$ gives UNKNOWN as a result. However, QueryPartition($h[A, B]$) $\bowtie$ QueryPartition($h[B, C]$) gives an 'upper bound' of it [71]. It is not a partition algebra join since the result is not a partition relation of h but it can be used to check consistency if the actual partition data of $h[A, B, C]$ is being imported as follows.

- If needed and available, $h[A, B, C; X]$ may be imported by ImportPartition. Consistency must be ensured with the existing data by projection and aggregation. Besides the above join as upped bound, checking of projection on the partition attributes is needed as well. Projection of new data on the previously known attribute combinations ensure the new data is compatible and covers the same base set. After the new data is imported, the logical level may be reorganized (some algebra operations attached to previously known nodes may be changed). A possible future extension using partial dependencies is to allow partial importing of a partition data set, i.e. the part that can be computed from existing data does not need to be imported.

6.3.5 Extending the schema with a derived attribute

Based on a partition $h[A, B]$ an attribute D is derived as follows:

1. In the first case, two attributes A, B are merged as D using a binary function f (relabeling function) defined on the symbol domain $\mathbb{S}$:

   DerivePartition($h[D], [A, B], (D = f(A, B))$)

   As a demonstrative example, assume the set names are strings for now. Function f can be the concatenation of the two strings and the constant '$\cap$' between them. This way the definition looks as $D = A\&' \cap' \&B$ and the resulting set names reflect the derivation semantics (intersection).

2. The second case demonstrates a case-switch. Value of D will be d_1 on a_1, d_2 on $b_1 \setminus a_1$ and d_3 elsewhere:

[8]The full partitioning $[A, B, C]$ may be computable for a part of h where $B \to C$ is valid. It is a future issue to extend the notion to allow local dependencies and determine the 'known' part and the 'unknown' part of $h[A, B, C]$ (see also derivation in [71]).

DerivePartition($h[D], [A, B]$,case($A = a_1 : D = d_1$; $B = b_1 \wedge A \neq a_1 : D = d_2$; $else : D = d_3$))

AB becomes a refinement of D and so $h[A, B, D]$ will also be known, i.e. value of GetKnownPartitions(h) becomes $\{[A, B, D]\}$. In both cases, DerivePartition calls the following to extend the structure with D:

DeclarePartition($h[A, B, D], \{AB \to D\}, \emptyset$)

Furthermore, it assigns the following algebra expression to node $[A, B, D]$ of the structural graph: $\sigma_{A=a_1}(R) \overrightarrow{\bowtie} \{(D : d_1)\} \uplus \sigma_{B=b_1 \wedge A \neq a_1}(R) \overrightarrow{\bowtie} \{(D : d_2)\} \uplus \sigma_{B \neq b_1 \wedge A \neq a_1}(R) \overrightarrow{\bowtie} \{(D : d_3)\}$, where R is the label of node $[A, B]$.

6.3.6 Selection query. Partitioning a subset

A selection (slicing) query can simply be issued by the extended syntax of QueryPartition, allowing a subsetting condition:

Rel =QueryPartition($h[A, B; X], (A = a \vee B = b)$)

To 'save' the base set corresponding the selection condition into the schema for future reference, a derived attribute should be defined:

DerivePartition($h[A, B], [K]$,case($A = a \vee B = b : K = k$; $else : K = \bar{k}$))

Set $h[K].k$ (abbreviated now as k) inherits attributes $[A, B; X]$. Assume table Rel' contains extra partitioning of the selected set (k) by a new attribute E with B and X. Incorporating it into the database can be done as follows (consistency with $h[B, K; X]$ is verified):

DeclarePartition($h[K].k$,PartStructCreate($[B, E; X], \emptyset, (X, +)$))
ImportPartition($h[K].k[B, E; X], Rel'$)

Querying partition E becomes possible on k or any of its subsets. Subset k and each of its subsets inherit A, B, X (as well as K). Subsets according to A and B inherit K and partition E becomes valid (not necessarily known) for their intersection with k.

- GetKnownPartitions($h[K].k$) – retrieves $\{[A, B, K; X], [B, E, K; X]\}$ (K is constant)
- QueryPartition($h[K].k[E]$) – partition E is valid and known on k,
- QueryPartition($h[K].k[B].b[E]$) – valid and known because k inherits B and $k[B].b$ inherits E,
- QueryPartition($h[B].b[K].k[E]$) – valid and known because b inherits K and that makes E valid for subset k of b (i.e. $b \cap k$),
- QueryPartition($h[K].k[A].a[E]$) – valid but unknown because k inherits A partition $[A, E]$ is not known,
- QueryPartition($h[A].a[K].k[E]$) – valid but unknown because a inherits K but partition $[A, E]$ is not known,
- QueryPartition($h[B].b[E]$) – is invalid because no E is declared for b. Subset $b \cap k$ must be selected as base set for E.[9]

[9]Note that in this example, $b \cap k = b$ so partition E may be computed for b but from the structural point of view it is out of scope, unless the model is capable to exploit local dependencies. If, for instance, DerivePartition were specified and implemented so that it creates local dependencies, $b[E]$ may be valid by observing the validity of $\emptyset \to K$ on b. It follows from the definition of the derived attribute K.

6.3.7 Navigation Operations handling partitioned subsets

A partitioning that is only valid for a subset allows special schema query and navigation operations to get such partitions from a superset. Recall the example from the previous subsection.

- $Sets$ =GetPartitionedSubsets(h) – retrieves $\{h[K].k\}$ because it is a base set with own (non-inherited) attributes according to the declaration of the structural level. Valid partitions can then be queried using GetPartStruct.
- χ =GetKnownSubPartitions($h[A,B,K]$) – retrieves $\{h[K].k[B,E;X]\}$ as the known partition of a subset of h as base set defined by one of the attributes A, B, K.
- $\tilde{\chi}$ =GetDirectSubRefinements($h[K;X]$) – retrieves $\{h[K].k[E;X]\}$ as a one-step refinement of partition K only available on a subset of h.
- χ =GetDirectSubRefinements($h[B,K;X]$) – retrieves $\{h[K].k[B,E;X]\}$ similarly.

A partition or set symbol must always be interpreted in the scope of a base set. However, each inherited partition has one or more ancestors (a partition with the same name of a superset) and there is a root ancestor (the originally defined partition with the same name and the largest base set). The base set of the root ancestor is the ***primary base set*** of an attribute. For a set symbol of a partition, the ***full extension*** is the subset with the same name in the root ancestor. These concepts can be used to query the 'origin' of a partition or one of its set symbols, useful for schema navigation from a subset towards the superset. For instance, the primary base set of partition E in this example is $h[K].k$ and it is the result of each of the following operations:

- k =GetPrimaryBaseSet($h[K].k[E]$)
- k =GetPrimaryBaseSet($h[K].k[B].b[E]$)
- k =GetPrimaryBaseSet($h[B].b[K].k[E]$)

The argument of GetPrimaryBaseSet must always refer to a singleton partition because the primary base set may not be unique otherwise.

Examples of full extensions:

- $h[B].b$ =GetFullExtension($h[A].a[B].b$)
- $h[K].k$ =GetFullExtension($h[B].b[K].k$)
- $h[B].b$ =GetFullExtension($h[K].k[B].b$)
- $h[B].b$ =GetFullExtension($h[K].k[E].e[B].b$)
- $h[K].k[E].e$ =GetFullExtension($h[K].k[B].b[E].e$)

6.3.8 Extension and unification of attributes

Assume a partition $h[Z, A, B, C]$ is given, with values of $h[Z]$ as $\{z_1, z_2, ...\}$. A partition D is given for subset z_1: $h[Z].z_1[A, B, D]$. Partition $h[F]$ can be constructed as an extension of D with the special value $\perp$ on $h \setminus z_1$:

ConstructPartition$(h[F], Z, case(Z = z_1 : F = z_1[D]; else : \perp))$

Furthermore, assume partition $h[Z].z_2[B, C, E, F]$ is also known. Based on this, partition $h[G]$ can be constructed as a unification of $z_1[D]$ and the merge of the two attributes $z_2[E, F]$ using a binary relabel function f:

ConstructPartition$(h[G], Z, case(Z = z_1 : G = z_1[D]; Z = z_2 : G = f(z_1[E], z_1[F]); else : \perp))$

ConstructPartition in its second parameter must always refer to one singleton partition as a determinant of the disjoint unification (now it is $h[Z]$). If such an attribute does not exist, it can be constructed by DerivePartition.

The effect of ConstructPartition on the logical level is an assignment of a disjoint union expression. The first step is to collect the inherited attributes from each branch and determine which combinations are known for each branch, together with those appear in the definition. The new attribute will be known together with these attribute combinations. In this second example, $h[B, Z, G]$ will be known only since $z_1[B, D]$ is known but not $z_1[C, D]$ and similarly, $z_2[B, E, F]$ is known but not $z_2[A, E, F]$. The attached operation will be the renamed disjoint union of the two expressions, with an extra operand for the else-branch: $h[B, Z, G] = \rho_{D\to G}(h[Z].z_1[B, Z, D]) \uplus \rho_{f(E,F)\to G}(h[Z].z_2[B, Z, E, F]) \uplus (\sigma_{Z\neq z_1 \wedge Z\neq z_2}(h[B, Z]) \overrightarrow{\bowtie} \{(\perp)\})$

6.3.9 Relationship of two sets. Characteristic partition

Assume partition $h[A, B]$ is known. The following examples are of set-theoretical operations for two sets $h[A].a$ and $h[B].b$:

- l =IsDisjoint$(h[A].a, h[B].b)$ – whether the sets are disjoint
- l =IsEmpty$(h[A].a[B].b)$ – whether a set is empty (now equivalent with the previous)
- $(l_{a\setminus b}, l_{a\cap b}, l_{b\setminus a}) = GetRelationship(h[A].a, h[B].b)$ – retrieves the complete set-theoretic relationship between the two sets indicating whether the differences and the intersection is empty

The implementation of these operations can be based on the partition relation of $h[A, B]$ (QueryPartition). If the common partitioning A, B were not known, the result may be unknown (but the aim is to implement the operations 'smart' so that they can take advantage of partial information – this is a future issue and closely related to local dependencies and horizontal decomposition).

Special cases of DerivePartition allow set-theoretic operations (by defining a new partition U in which subset u is constructed and $\bar{u}$ is the complement of it):

- ConstructUnionPartition$(h[U], u, \bar{u}, h[A].a, h[B].b)$ – results $u = a \cup b$
- ConstructIntersectPartition$(h[U], u, \bar{u}, h[A].a, h[B].b)$ – results $u = a \cap b$
- ConstructDiffPartition$(h[U], u, \bar{u}, h[A].a, h[B].b)$ – results $u = a \setminus b$

- ConstructSimmDiffPartition$(h[U], u, \bar{u}, h[A].a, h[B].b)$ – results $u = a\Delta b$

For example, the attribute definition argument of the union, passed to DerivePartition is $case(A = a \vee B = b : U = u;\ else\, U = \bar{u})$. Others follow similarly.

Generalization of these operations for two arbitrary sets in the database may be possible using the concept of characteristic partition, described below.

6.3.10 On the way of relating two arbitrary sets

This last case sketches how two arbitrary sets may be related and a partition of their union constructed. This procedure usually needs external information (some partition combinations introduced on the structural level that need to be 'known', i.e. populated with data). In some cases, the available information may be enough if horizontal separation can be considered. To automatize this, a 'smart' deduction system is needed to assign expressions to nodes on the logical level that can take advantage of the available information. This, again, points towards the future exploitation of local functional dependencies. This case, therefore should be considered as a future prospect.

Relationship and common partition of sets can be examined in terms of partitions of a common superset. With set path expression and attribute structure, a common superset of two arbitrary sets can be computed (it is not necessarily the smallest common superset – the aim is to develop methods that compute a 'good' common superset based on the available information).

Assume the schema of Section 6.3.9 is extended with $h[A].a[E]$ and $h[B].b[F]$, with two subsets e and f picked from these partitions, respectively. Their common superset is h:

$$h = GetCommonSuperset(h[A].a[E].e, h[B].b[F].f)$$

To make the two sets comparable, a 'characteristic partition' is constructed for both of them in h: Characteristic partition $h[\tilde{E}]$ of e is constructed with two values e and $\bar{e}$, denoting $h \cap e$ and $h \setminus e$, respectively. Partition $h[\tilde{F}]$ is constructed the similar way:

ConstructCharacteristicPartition$(h[\tilde{E}], e, \bar{e}, h[A].a[E].e)$
ConstructCharacteristicPartition$(h[\tilde{F}], f, \bar{f}, h[B].b[F].f)$

This operation calls ConstructPartition once or several times, according to the attribute base set assignment structure (possibly nested).

Theoretically, relationship of the two sets may now be queried or new sets may be constructed out of them by the operations in the previous subsection, based on the common partitioning $h[\tilde{E}, \tilde{F}]$. However, in most cases this common partition will not be known immediately.

Set b (and f) inherits partition $\tilde{E}$ but the known partitions by default are $b[A, B, F]$ and $b[A, B, \tilde{E}]$ if no global dependency is valid for the new attributes (usually not). These can not always be merged without external information if $b[F, \tilde{E}]$ cannot be computed (it corresponds to $e \cap f$). However there are cases when it can be computed without a global dependency, e.g. when e and f are disjoint (or one covers another) and $F \to \tilde{E}$ is valid for one part of b and $\tilde{E} \to F$ is valid for its complement. We face the same situation when a inherits the characteristic partition $\tilde{F}$ of f. Horizontal separation in the partition model needs to be studied in detail.

The case becomes even more complicated if even $h[A, B]$ is not a known partition, but easier if $A = B$.

If the two characteristic partitions are comparable then the union of two sets can be constructed and ConstructPartition can be used to build a common partition out of their own partitions. The assignment of relations on the logical level has again a key role, in order to get the maximal known common partitions with the newly constructed partition. In the ideal case, a smart deduction mechanism is extended with user input facilities, as the user specifies what partition (s)he wants to get as known, and the system asks only for the necessary external information that does not follow from the available data. This can be an aim of an improved partition model.

6.4 Evaluation against the Motivating Cases

The sample cases above illustrate how some of the situations in Section 3.4 can be described in the partition model.

Partially applicable attributes (Section 3.4.1) can be modeled by the attribute base set structure. A base set of an attribute can be an arbitrary set described by a subset path (SetPath value) starting from the global base set. If a set cannot be expressed by a simple subset path then a derived attribute can be defined, merging one or more subsets. The merged set can be referenced by a single set name (symbol) of a partition relation.

On the structural level each attribute must be specified for a base set. The meaning of this declaration is to fix the possible partitions. Partially applicable combinations (Section 3.4.2) is possible by known-unknown partitions. However, the framework does not take full advantage of horizontal decomposition and more sophisticated derivation techniques. This should be improved further in order to get more known partitions given some partial information.

Complex dimensional structures presented in Section 3.4.3 can be modeled using functional dependencies. Dimensions may either be implicit (implied by independence in terms of functional constraints) or explicit (as concepts modeled by namespaces). Ad-hoc correspondence of heterogenous dimension levels can be achieved using (possibly temporary) derived attributes. In order to capture some of the inhomogenous hierarchies, the model should be improved to exploit horizontal separation facilities, local dependencies.

Inhomogenity of aggregation (Section 3.4.4) is only partially captured. Aggregation is restricted to cumulative (distributive) attributes but Section 6.5.5 will sketch some of the future prospects. An aggregation which is not available for all granularities can be modeled by unknown partitions with cumulative attributes. The partition model constructs new sets always by unifying disjoint sets, thus ensuring summarizability. This way the aggregation values for a coarser partition are automatically computed if they are given given for a finer partition. Some aggregations may be known up to a specific level of detail. Importing new information into the database includes aggregation consistency check in terms of the robust, well-defined partition structure on the structural level.

Other issues such as distributed storage (see Section 3.4.5) are not discussed in the partition framework yet. The logical level may allow remote data sources or on-demand 'stubs' and the concept of known partitioning improved in the future with horizontal separation allows to minimize the need of data transfer by deriving all information possible from the locally available data. If a dataset is downloaded from a remote source then it can be incorporated as a partition becoming known. However, a proper data manipulation framework is needed to keep all data up-to-date.

Various representations may be allowed on the interpretation level. For instance, an overlay of geographical fields (Section 3.1) can be modeled as a partition schema and the atomic set intersections may correspond to the elementary polygons of a polygon topology. Much of the operations may be performed outside the interpretation level by the partition framework. Simultaneous handling of various representations depends on the further development of the interpretation level.

The partition framework may be improved to get a general set-based meta-database system over various data assets (Section 3.4.7). It is a possible direction of future work.

6.5 Future Issues

The formalism presented here can be used as a basis for developing an improved partition database framework. There are many open issues – some of them are presented here briefly. The main tasks include surveying possible applications and modeling real-life scenarios based on the cases in Section 3.4 and improve the framework according to the problems arise.

6.5.1 Complexity, Consistency

Complexity, especially on the logical level can be rather high. The size of the structural graph can be exponential in the number of partition attributes. Assigning and verifying partition algebra expressions one-by-one is infeasible. It must be supported by semi-automatic methods ('smart' deduction) based on normalization and view selection theory. Representation of the graph and elements of the structural level is another issue of complexity. It is likely that the full graph has not to be stored, only some of its 'critical' nodes.

There are many cases when a consistency check is needed. A precise consistency requirements analysis is needed for each level, always against the upper level – similarly to the relational theory: an instance satisfies the schema constraints if it obeys certain conditions. Complexity of full consistency check is high, since a join expression must be computed to verify a partition of a previously unknown attribute combination. Moreover, aggregation consistency needs computing and verifying the values for each cell and exact equality can only be expected if data comes from the same source.

Furthermore, it is a natural need to allow the attribute names not being globally unique. Proper name conflict handling must be ensured in such case.

6.5.2 Expressivity and scalability

As introduced in Section 6.1.1, the model can be customized by extending or replacing some parts of the language. These parts are called plug-ins. A solid foundation is needed for the specification of plug-ins and their compatibility and impact on consistency. Expressivity of the language depends on the chosen plug-ins.

Assuming a structure of the set names is a promising direction, especially the point of compound set names and reflecting semantics in set names. This may lead to more sophisticated user interface and aggregation handling. Set and partition definition expressions may use advanced construction or selection operations and exploit the facilities of a set symbol algebra. Aggregation extension is considered in a separate subsection.

6.5.3 Theoretical foundations

This chapter together with 4 and 5 can be used as a basis for the complete formal foundation. The partition model can be viewed as a combination of the semi-structured and the relational data model.

Concept of completeness must be defined for the set of operations in order to study the completeness of partition operations. Effects of operations must be studied in more detail, as well as methods for their implementation details, especially to make a complete realization of the effects of structural level operations on the logical level (automatic assignment of partition algebra expressions by implication, 'smart' deduction, including reorganization of the schema if needed).

Schema evolution may include declaring an occasional property of data as a constraint or release a constraint based on new data. This must be effectively supported by further operations. Expressive power of an improved partition language must be surveyed.

The concepts related to namespaces are not fully developed yet (types of namespaces, nesting). Dimensions can be obtained from the set of refinement (functional) constraints. An explicit dimension concept should be based on both namespaces and constraints Another role of namespaces is scope handling and persistency. Regulations for namespaces (derivation, etc.) and ensuring their consistency is another question that needs future work.

To formally prove completeness of structural graph notation extended with subsetting and equivalence edges is an important task, as well as the implication of base set constraints or the derivation of them from the graph. The base set constraint notation probably needs extension in order to describe more complex relationships among base sets.

The concept of horizontal decomposition is a key issue in future improvement to help taking advantage of all available information (including local constraints) for derivation of partition relations of subsets. Attribute unification can simulate horizontal decomposition but a more natural way is necessary.

6.5.4 Error handling for missing information

The structural level allows queries in terms of partition attributes and not driven by the available information (relational tables). There are cases when the answer is UNKNOWN. To improve this behavior a smart answering method may be developed: If a query cannot be answered then the system would answer with a partition schema required for the computation of the query. This is usually not unique but the aim is to get the smallest schema (small base set a few attributes) that is enough (e.g. the common partition according to some attributes is not known for a subset).

This may be used in a peer-to-peer system where each network node has partial knowledge and the communication costs are to be minimized. If a user queries a partition which is not known locally, the system may find the needed information on-demand and call ImportPartition.

6.5.5 Aggregations

Aggregation facilities must be extended beyond cumulative attributes. To ensure correct aggregations (summarizability), aggregation semantics must be introduced for partition semantics. This will hold how an aggregation attribute can be computed from the population data. Since a partition database usually does not contain item-level data, aggregation handling can be based on symbolic formula-manipulation, aggregation values of certain subsets and auxilliary data [59].

Basic types of aggregation are stock, flow, and value/item (Section 3.3). The partition framework naturally models flow-type aggregations with temporal partition attributes. With respect to summarizability, the common terms distributive, algebraic and holistic may be refined according to a semantical approach: during operations with partitions, aggregation semantics is 'carried' together with the values and the question is what extra data needs to be carried for sets in order to compute the aggregation value of their disjoint union. This is the computational aspect of aggregation.

The conceptual aspect is whether an aggregation is meaningful. Difference between temporal and non-temporal aggregation can be generalized as time becomes an ordinary partition. If a temporal summarizability is invalid, it usually means the aggregation refers to objects but the population consists of object-time pairs. This should be made clear on the conceptual level with the concept of partitions. Moreover, spatial summarizability may face similar problems as temporal summarizability, due to the continuous nature of geographical space.

An algebraic aggregation attribute can be defined as a derived attribute. If an aggregation is performed, the new values are computed automatically, using the attached derivation formula. If X is a sum of some values for each subsets of a partition and N stores the number of items in each set, the average $\bar{X}$ can be defined as follows:

$$\text{DeriveAggregation}(h[A, B; X, N], \bar{X}, (X/N))$$

It is up to future work to survey all the effects and the allowed operations of such a facility. The result can be a plug-in for algebraic aggregation definition expressions.

An important feature should be to get aggregation values from partition attributes. For instance, if the user projects out a partition attribute A then a new aggregation value may be defined, representing the number of different values of A for the resulting subsets (how many of the component sets of A intersects a resulting subset). This is not a cumulative attribute and to provide correct summarization, auxiliary information is needed. It may also depend on the attribute refinement structure. The reverse direction is to create partitions according to aggregation values. It may need even more complex semantical constructs.

If an average of sums is computed, it may be called 'compound aggregation'. Handling such cases should be based on introducing another partition level. The sum is computed over a population in terms of a partition. Subsets of this partition should be treated as items of the second-level population and the average of sums can be interpreted in terms of a partition over them. By developing such a multilevel partition framework, aggregation semantics can be tracked properly while the user performs ad-hoc analytic queries.

The aim is to have a description language that captures partition and aggregation semantics, 'carries' this semantics together with the data and transforms it properly when performing operations. If the semantics cannot be transformed in a meaningful way, the operation should not be allowed. Partition and aggregation semantics may be used for developing the concepts needed for 'fair' comparisons as well [60, 59].

6.5.6 Data representation

Although the partition model is based on the relational database theory due to its convenience in set intersection reference by tuples, it is not necessary to store partition data in a relational database. For instance, multidimensional data can be stored effectively using using MOLAP technology. The partition model can be treated as a database structuring, metadata or data access language for data warehouses, based on semantics.

Values of a derived attribute may be stored for query efficiency or computed on-demand by its derivation formula. Constraint databases [66] allow representing data as equation and unequality formulae. In some cases the set structure can be effectively captured by a constraint formula, e.g. in a time dimension each day is similarly divided into 24 hours and each hour divided into 60 minutes. This can be stored as a formula. Constraints allow a finite representation of infinite sets (e.g. point sets in spatial databases or time intervals in temporal databases). Representation may also be heterogenous and the upper levels of the partition database provide unified access.

The database must be self-contained, i.e. it should store the structural level information as well, derivation of attributes, base set relationships, etc. Such structures form directed acyclic graphs. To describe the structural level schema, an XML syntax shall be defined. This can be combined with traditional relational tables and view definition on the logical and instance levels.

6.5.7 Data manipulation

The defined basic partition language operations can be used to construct and query the database. In order to make a partition database work, data manipulation must be considered. This should not be based on insertion and deletion of rows in the partition relations. Data manipulation operations must rather be defined based on partition semantics. Basically two causes exist of data manipulation. The first is the change in the population having an effect on the partition database. Such case arises if the population is extended with new items, for instance. Definition of partitions do not change but must be updated as necessary. The second case is when the population does not change but the partition structure is modified or redefined.

This topic raises many questions and needs further investigations. Only some needs and starting ideas are sketched below.

Extension of the population

This is the most likely data manipulation need in a typical data warehouse for archiving historical data. To handle an extension of the population, an extra partition attribute can be defined indicating the new elements. If the database has active connection to the population-level data, the partition definitions are available and they can be applied to the new data, this extension can automatically performed. Exceptions can always occur and the new elements that cannot be classified according to the existing rules must be collected. Some refinement or disjointness constraints may be violated by new data and must be revised.

Modifying the partition schema

Data manipulation may be needed in cases even if the population does not change. The existing schema construction operations allow incorporating new knowledge into the schema. Changing of the schema needs further operations, e.g. to redefine a partition. Such a redefinition has 'side effects' in the schema, which must be revealed.

If a new partition is defined that is not consistent with the existing data, either the new or the existing data is wrong. There should be a way to insert data in 'force' mode, i.e. modifying the schema to be consistent with the new information. There is usually no unique way for handling such cases so the system must allow the user to choose among some possible modifications.

Part III

A Contribution to the Theory of Functional Constraints

Chapter 7

Notation and Axiomatization of Functional Constraints

This chapter presents a simplified syntax and suitable axiomatization for functional dependencies and negated functional dependencies. It is the formal foundation of the spreadsheet and graphical representation for sets of constraints and reasoning framework (see the subsequent chapters) as well as of the generation and computation of the number of constraint sets (see also [32, 34, 31]). Functional dependencies are used in the partition model to describe the refinement structure of partition attributes (Chapter 5).

In general, use of the ST axiomatization (Section 7.3) is proposed instead of the traditional Armstrong system, due to the simplicity of rules and the order of rule application.

7.1 Notation

7.1.1 Universes of Constraints

Treating sets of functional constraints becomes simpler if we avoid dealing with obviously redundant constraints. Consider two well-known special types of constraints first. A *canonical (singleton)* functional dependency or a singleton excluded functional constraint has exactly one attribute on its right-hand side. A *trivial* constraint (a functional dependency or an excluded functional constraint) is a constraint with at least one attribute of its left-hand side and right-hand side in common or has the empty set as its right-hand side. It will be shown that we do not loose relevant deductive power if we represent non-trivial canonical functional dependencies and non-trivial singleton excluded functional constraints only. This chapter introduces a formal system in Section 7.3 corresponding to the restriction (simplification) to the universe of functional constraints, allowing the derivation of all non-trivial, singleton implications of a set of constraints without the need to include any non-singleton or trivial constraints in the proofs. This provides a reasoning facility in terms of the simplified representation.

Notations $\mathbb{D}$, $\mathbb{D}^+$ and $\mathbb{D}_c^+$ stand for the universes of functional dependencies, non-trivial functional dependencies and non-trivial canonical (singleton) functional dependencies, respectively, over a fixed underlying domain of attribute symbols. Similarly, $\mathbb{E}$, $\mathbb{E}^+$ and $\mathbb{E}_c^+$ denote the universes of excluded functional constraints, non-trivial excluded constraints and non-trivial singleton excluded functional constraints (negated non-trivial, canonical dependencies) over the same set of attribute symbols, respectively. The traditional universe of functional constraints (including functional dependencies and excluded constraints) is $\mathbb{D} \cup \mathbb{E}$ while the simplified representation deal with sets of constraints over $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$.

In most cases, the focus is on *closed* sets of functional dependencies. The original definition of closure in Section 2.3 is now adapted to match the restricted syntax: A finite set $\mathcal{F} \subset \mathbb{D}_c^+$ is closed if and only if $\mathcal{F}^+ = \mathcal{F}$ where $\mathcal{F}^+$ is the closure of $\mathcal{F}$, i.e. $\mathcal{F}^+ = \{\delta \in \mathbb{D}_c^+ \mid \mathcal{F} \models \delta\}$. Note that by excluding the trivial constraints from the system, $\mathcal{F}^+$ contains constraints referring to the attributes occuring in the constraints of $\mathcal{F}$ only, so the closure becomes independent of the other attributes of the schema.

7.1.2 The Notion of Dimension

For the classification of functional constraints and the attributes they refer to, the notion of dimension is introduced. Dimension of a constraint is simply the size of its left-hand side, i.e. the number of attributes on its left-hand side. Note that after getting rid of the non-singleton constraints, each constraint we treat has exactly one attribute on its right-hand side. Since we also excluded the trivial constraints from our system, the left-hand side and the right-hand side of a constraint are always disjoint. Therefore, it makes sense for a specific attribute to consider the minimal determinant of it, i.e. the left-hand side of a constraint with that attribute on its right-hand side and its left-hand side minimal. The size of this minimal determinant is defined as the dimension of the attribute. We put these definitions to a more precise form.

For a functional dependency $X \to A \in \mathbb{D}_c^+$ denote by $[X \to A]$ its *dimension*, defined as

$$[X \to A] \overset{\text{def}}{=} |X|$$

(dimension of an excluded functional constraint can be defined similarly). For a single attribute A, given a set of functional dependencies $\mathcal{F} \subset \mathbb{D}_c^+$, *dimension of* A is denoted by $[A]_{\mathcal{F}}$ (or just simply $[A]$) and defined as

$$[A]_{\mathcal{F}} \overset{\text{def}}{=} \min_{X \to A \in \mathcal{F}^+} |X|$$

This definition is extended with $[A]_{\mathcal{F}} \overset{\text{def}}{=} \infty$ for the case when no $X \to A$ exists in $\mathcal{F}^+$ [1].

Closed sets of functional dependencies (and the corresponding relationship types) can be classified according to the dimensions of the attributes (see Chapter 8). As will be seen later in Chapter 9, this notion of dimension closely relates to the triangular graphical representation.

7.2 Principles of Simplified Reasoning

The universe of the extended Armstrong implication system (see Section 2.3.3) is $\mathbb{D} \cup \mathbb{E}$ (functional dependencies and excluded functional constraints over a set of attributes). From the axiomatic point of view, a reduction to sets of constraints over $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$ (singleton, non-trivial constraints) can be achieved by two steps.

A natural simplification to handle sets of constraints without loosing any relevant information is omitting trivial constraints and non-singleton functional dependencies since trivial dependencies always hold (trivial excluded constraints must always be false, consequently) and non-canonical functional dependencies can always be substituted by canonical dependencies[2]. However, the axiom and rules of the extended Armstrong

[1] An alternative definition would be $[A] = n$ (where n is the number of attributes considered) if no $X \to A$ exists. However, the notation ∞ emphasizes that A is independent of other attributes and makes the definition independent of n.

[2] For example, $X \to AB$ is represented as $X \to A$ and $X \to B$. Excluded functional constraints with more than one attribute on their right-hand sides can not be eliminated this way. However, we show that omitting these can also be achieved.

implication system do not allow reasoning over the simplified syntax. It will be shown that an equivalent implication system can be constructed if these restrictions are applied to the universe of constraints (systems ST and NST in Section 7.4). It turns out that no trivial dependencies are needed in the new system to deduce all non-trivial consequences of a given set of (positive and/or negative) constraints. That is the reason we can "forget" about trivial dependencies and trivial negated constraints and completely exclude them from the formal system.

The second step is omitting the non-singleton excluded functional constraints. Although this is a real restriction, non-singleton excluded functional constraints are not needed for describing (the complement of) a closed set of functional dependencies since they are interpreted as disjunctions.[3] For a closed set, it is exactly known which of them holds. Deriving the full knowledge starting with an initial set of constraints may result some non-singleton excluded constraints. However, their relevance is rather low if they can not be simplified.[4] Neither is it likely for an initial set to contain this kind of excluded constraints. It will be shown by constructing another formal system that the non-singleton excluded constraints are not needed for deriving a singleton excluded constraint when the initial set contains only singleton constraints.

The first step mentioned above can be formalized as restricting the universe to $\mathbb{D}_c^+ \cup \mathbb{E}^+$ and for each finite subset $\mathcal{F}$ of $\mathbb{D} \cup \mathbb{E}$ the set $\mathcal{F}' = \{V \to C \mid \exists W : V \to W \in \mathcal{F}, C \in W \setminus V\} \cup \{V \not\to U \mid \exists W : V \not\to W \in \mathcal{F}, U = W \setminus V\}$ is an equivalent representation over $\mathbb{D}_c^+ \cup \mathbb{E}^+$. The second step is taking $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$ as the universe and disallowing constraints in the form $V \not\to W$, $|W| > 1$ to be elements of the initial set.

However, the extended Armstrong implication system makes it possible to derive constraints outside $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$ even if the initial set contains only constraints over this universe, and it must be guaranteed these intermediate results are not necessary for deducing some of the consequences belonging to $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$. Completeness of implication is achieved by an alternative axiomatization called PQRST Implication System. The rules will also have the advantage that each of them refers to only one set of attributes and the rest is of single, different attributes not occuring in the set. This way it becomes much easier to recognize a possible instantiation of the rules in an attribute-based representation (as opposed to a subset-based representation).

It also turns out there exists a specific order of application of the rules, which leads to a complete deduction algorithm for elicitation of the full knowledge a set of constraints holds.

7.3 The Main Ruleset

7.3.1 The ST and PQRST Implication Systems

In each of the following rules and axioms, letter Y denotes a set of attributes (allowed to be empty), while A, B and C stand for different single attributes not occuring in Y.

$$\text{(S)}\ \frac{Y \to B}{YC \to B} \qquad \text{(T)}\ \frac{Y \to A,\, YA \to B}{Y \to B}$$

$$\text{(P)}\ \frac{YC \not\to B}{Y \not\to B} \qquad \text{(Q)}\ \frac{Y \to A,\, Y \not\to B}{YA \not\to B}$$

$$\text{(R)}\ \frac{YA \to B,\, Y \not\to B}{Y \not\to A} \qquad (\square)\ \neg(Y \to B,\, Y \not\to B)$$

[3]For instance, $X \not\to AB$ means either $X \not\to A$ or $X \not\to B$ (or both) must hold.

[4]If neither $X \to A$ nor $X \to B$ can be deduced then $X \not\to AB$ can not be simplified and one just conclude the lack of information: it is not possible to decide whether the dependencies hold or not. Using the closed world assumption [2] as a possible model, one might finally conclude neither of the two dependencies hold. Otherwise, we state the lack of information and as soon as one of the dependencies is determined, a new reasoning session starts with the extended initial set so the non-singleton negated constraint will not be needed.

Restriction of the universe to singleton, non-trivial constraints is possible from the syntactic point of view because none of the implication rules presented above can be used to deduce trivial or non-singleton functional dependencies if the initial set contains only non-trivial and singleton dependencies. The same holds for excluded functional constraints.

These rules can directly be applied for deducing consequences of a set of constraints given in terms of the graphical or spreadsheet representation, as it will be demonstrated in Section 9.3 and Section 8.3. No explicit rule for transitivity is needed, [31] shows an example on how transitivity can be simulated with these rules.

Rules (P), (Q) and (R) act as negations of (S) and (T) for deducing excluded functional constraints. ($\Box$) is a formalization of '$\not\rightarrow$' being the negation of '$\rightarrow$', i.e. $\neg(\Box)$ can be deduced starting with a contradictory set of constraints.

Two formal systems can be defined with the above rules as follows:

- the *ST implication system* over $\mathbb{D}_c^+$ with rules (S) and (T) and no axioms,
- the *PQRST implication system* over $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$with all the presented rules and the symbolic axiom ($\Box$), which is used for indicating contradiction.

These systems are sound and complete for deducing non-trivial, singleton constraints. This statement together with others mentioned above is formalized by the following two theorems:

Theorem 1 The ST system is sound and complete over $\mathbb{D}_c^+$, i.e. $\mathcal{F} \vdash_{ST} \delta \iff \mathcal{F} \models \delta$ for each finite subset $\mathcal{F}$ of $\mathbb{D}_c^+$ and $\delta \in \mathbb{D}_c^+$.

Theorem 2 Let $\mathcal{F}$ be a finite subset of $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$ and $\delta \in \mathbb{D}_c^+ \cup \mathbb{E}_c^+$.

The PQRST system without ($\Box$) is sound over $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$ and complete with the restriction that $\mathcal{F}$ cannot be contradictory, i.e. $\mathcal{F} \vdash_{PQRST} \delta \iff \mathcal{F} \models \delta$ for each non-contradictory $\mathcal{F}$. Moreover, $\neg(\Box)$ can be derived if and only if $\mathcal{F}$ is contradictory.[5]

7.3.2 Order of Rule Application

The implication systems introduced above have the advantage of the existence of a specific order of rule application providing a complete algorithmic method for getting all the implied functional dependencies and excluded functional constraints starting with an initial set.

It is based on the following theorem:

Theorem 3

1. Let $\mathcal{F}$ and $\mathcal{G}$ be finite subsets of $\mathbb{D}_c^+$. If $\mathcal{F} \vdash_{ST} \mathcal{G}$ then all elements of $\mathcal{G}$ can be deduced starting with $\mathcal{F}$ by using the rules (S) and (T) the way that no application of (T) precede any application of (S).

[5]Contradictory means two opposite constraints (e.g. $X \rightarrow W$, $X \not\rightarrow W$) can be derived using the extended Armstrong implication system (they might be trivial as well). Showing that $\neg(\Box)$ (with non-trivial, singleton dependencies) can be deduced using PQRST is satisfiable in such cases. Contradictory cases in the Armstrong system allow deduction of some non-trivial negated constraints which can not be derived using PQRST the same way because trivial constraints are not allowed (e.g. one can get $X \not\rightarrow \emptyset$ with the extended Armstrong system using rule (6) first and then for any B, $X \not\rightarrow B$ follows using rule (4)). Although these cases are irrelevant, formal completeness can not be stated rigorously.

2. If $\mathcal{F}$ and $\mathcal{G}$ are finite subsets of $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$ and $\mathcal{F} \vdash_{PQRST} \mathcal{G}$ then all elements of $\mathcal{G}$ can be deduced starting with $\mathcal{F}$ by using the rules (S), (T), (R), (P) and (Q) the way that no application of (T) precede any application of (S), no application of (R) precede any application of (T) and no application of (P) or (Q) precede any application of (R). Order of (P) and (Q) is arbitrary. Furthermore, (R) is needed to be applied at most once if $|\mathcal{G}| = 1$.

The order of rule application can be formulated as algorithmic methods. The method for negated constraints is an extension of the method for positive constraints.

The ST Algorithm for Sets of Functional Dependencies

Considering the implication system ST, Part 1 of Theorem 3 yields the following derivation algorithm:

1. Starting with the given initial set of non-trivial, canonical functional dependencies as input,
2. extend the determinants of each dependency using rule (S) as many times as possible, then
3. apply rule (T) until no changes occur.
4. Output the generated set.

The form of rules allow to fine-tune this algorithm by dimensionality: start with low dimensional constraints and apply rule (S) towards higher dimensions. When completed, start with the highest possible dimension class and apply rule (T) towards lower dimensions.

The STRPQ Algorithm for Sets of both Positive and Negative Constraints

Incorporating excluded functional constraints into the system, rules (P), (Q) and (R) can be applied as complements of rules (S) and (T). This gives the following algorithm called *STRPQ algorithm* (see Part 2 of Theorem 3):

1. Starting with the given initial set of non-trivial, singleton functional dependencies and excluded functional constraints input,
2. extend the determinants of each dependency using rule (S) as many times as possible, then
3. apply rule (T) until no changes occur,
4. apply rule (R) until no changes occur,
5. reduce and extend the determinants of the excluded constraints using rules (P) and (Q) as many times as possible.
6. Output the generated set.

7.4 Implication Systems with Non-Singleton Excluded Constraints

Proofs of Theorems 1 and 2 are based on the soundness and completeness of the Armstrong and extended Armstrong implication systems over $\mathbb{D}$ and $\mathbb{D} \cup \mathbb{E}$, respectively (proofs will follow in Section 7.5). Rules (S), (T), (P), (Q) and (R) can be used for implications over $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$. As an intermediate step, the universe is extended with non-singleton negated constraints ($\mathbb{D}_c^+ \cup \mathbb{E}^+$) and introduce another system called NST in this subsection. It will be shown that the NST system is equivalent with the system PQRST over the original, restricted universe ($\mathbb{D}_c^+ \cup \mathbb{E}_c^+$). Two other systems called U and UE are introduced afterwards, which are equivalent to the ST and NST systems, respectively. Systems U and UE will then be compared to the (extended) Armstrong axiomatization which leads to the completion of the proofs of theorems.

The universe is now extended to $\mathbb{D}_c^+ \cup \mathbb{E}^+$, i.e. non-singleton excluded functional constraints are allowed.

7.4.1 The NST Implication System as an Extension to the PQRST System

In each of the following rules and axioms, letters Y, Z and W denote pairwise disjoint sets of attributes (the sets are allowed to be empty with the restriction that right-hand sides of excluded constraints can not be empty), while A, B and C stand for different single attributes not occuring in the sets of the same rule.

$$\text{(S)}\ \frac{Y \to B}{YC \to B} \qquad \text{(T)}\ \frac{Y \to A,\, YA \to B}{Y \to B}$$

$$\text{(NS)}\ \frac{YC \not\to Z}{Y \not\to Z} \qquad \text{(NT1)}\ \frac{Y \to A,\, Y \not\to Z}{YA \not\to Z} \qquad (*)\frac{Y \not\to Z}{Y \not\to ZC}$$

$$\text{(NT2)}\ \frac{YZW \to B,\, Y \not\to ZB}{Y \not\to ZW} \qquad (\square)\ \neg(Y \to B,\, Y \not\to B)$$

$$\text{(NT2S)}\ \frac{Y \to B,\, Y \not\to ZB}{Y \not\to Z}$$

For deriving functional dependencies (positive constraints), rules ((S) and (T)) are the same as before. For deducing excluded functional constraints, rules (NS), (NT1), (NT2) and ($*$) act as negations. Although (NT2S) can be derived from (S) and (NT2) with $W = \emptyset$ (see Lemma 8 later), it is included separately since this case may often arise.

In practice, ($*$) is not needed since all relevant information for excluded constraints can be derived without it, as it will be shown (see Theorem 4, part 2). It means we do not need to extend the right-hand sides of the negated constraints during elicitation of the full knowledge on valid constraints. This is important since negated dependencies with more than one attribute on their right-hand sides are interpreted as disjunctions and are not represented directly in the graph or spreadsheet and our goal is usually to reduce their right-hand sides. Using (NT2) remains the only case that can increase the size of the right-hand side of a negated constraint but with removing at least one attribute B at the same time ($B \notin W$).

The *NST implication system* over $\mathbb{D}_c^+ \cup \mathbb{E}^+$ is defined as the set of rules just presented ((NT2S) is optional), including the symbolic axiom ($\square$), which is used for indicating contradiction. This system is sound and complete for deducing non-trivial constraints. Denote by *NST'* the formal system NST with ($*$) excluded.

Theorem 4 Let $\mathcal{F}$ be a finite subset of $\mathbb{D}_c^+ \cup \mathbb{E}^+$ and $\delta \in \mathbb{D}_c^+ \cup \mathbb{E}^+$.

1. The NST system without ($\square$) is sound over $\mathbb{D}_c^+ \cup \mathbb{E}^+$and complete with the restriction that $\mathcal{F}$ cannot be contradictory, i.e. $\mathcal{F} \vdash_{NST} \delta \iff \mathcal{F} \Vdash \delta$ for each non-contradictory $\mathcal{F}$. Moreover, $\neg(\square)$ can be derived if and only if $\mathcal{F}$ is contradictory.

2. Using rule $(*)$ is not necessary for deducing all relevant information with NST, i.e. the NST' system is still sound over $\mathbb{D}_c^+ \cup \mathbb{E}^+$, complete over $\mathbb{D}_c^+$ and if $\mathcal{F}$ is not contradictory and $\delta \in \mathbb{E}^+$, $\delta = X \nrightarrow Y$, then $\mathcal{F} \Vdash \delta \Longrightarrow \exists Z \subseteq Y, Y \neq \emptyset : \mathcal{F} \vdash_{NST'} X \nrightarrow Z$.[6] If $\mathcal{F}$ is contradictory then $\mathcal{F} \vdash_{NST'} \neg(\Box)$ holds.

Proof of Theorem 2 is based on Theorem 4 and the following lemmas stating the equivalence of systems PQRST and NST over $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$.

Lemma 1

If NST is sound then PQRST is sound, i.e. $\vdash_{NST}\{$(P),(Q),(R)$\}$.

Lemma 2

Starting with a set of non-trivial, singleton (positive or negative) constraints each non-trivial, singleton constraint that can be deduced using the NST' system over $\mathbb{D}_c^+ \cup \mathbb{E}^+$ can also be deduced using the PQRST system over $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$ if the initial set is not contradictory. For a contradictory set over $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$, $\neg(\Box)$ can be deduced using PQRST. More precisely,

- let $\mathcal{F}$ be a finite subset of $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$, X a set of attributes and A a single attribute such that $A \notin X$. If $\mathcal{F}$ is not contradictory and $\mathcal{F} \vdash_{NST'} X \nrightarrow A$, then $\mathcal{F} \vdash_{PQRST} X \nrightarrow A$. For a contradictory set $\mathcal{F}$, $\mathcal{F} \vdash_{PQRST} \neg(\Box)$ holds.
- Moreover, if $\mathcal{F} \vdash_{PQRST} X \nrightarrow A$ holds, then deduction can be performed so that rule (R) is used at most once.

7.4.2 The U and UE Implication Systems

Consider the following rules. Like before, letters X, Y, Z and W denote pairwise disjoint sets of attributes for each rule they occur in (they are allowed to be empty with the restriction that right-hand sides of excluded constraints can not be empty) while A_i's ($i \in [1..k]$, $k \in \mathbb{N}_0$) refer to distinct attributes not occuring in any of the attribute sets of the same rule.

(U) $\frac{XY \rightarrow A_1, \cdots, XY \rightarrow A_k,\, YA_1 \cdots A_k \rightarrow B}{XY \rightarrow B}$

(E1) $\frac{XY \rightarrow A_1, \cdots, XY \rightarrow A_k,\, XY \nrightarrow ZA_1 \cdots A_l \ (0 \leq l \leq k)}{YA_1 \cdots A_k \nrightarrow Z}$

(E2) $\frac{X \rightarrow A_1, \cdots, X \rightarrow A_k,\, X \nrightarrow ZA_1 \cdots A_k}{X \nrightarrow Z}$

(E3) $\frac{XZW \rightarrow A_1, \cdots, XZW \rightarrow A_k,\, XY \nrightarrow ZA_1 \cdots A_k}{XY \nrightarrow ZW}$

(E$\Box$) $\neg(X \rightarrow A_1, \cdots, X \rightarrow A_k,\, X \nrightarrow A_1 \cdots A_k)$

Let us call the system of $\{$(U)$\}$ over $\mathbb{D}_c^+$ *U implication system.* Extended by (E1), (E2), (E3) and (E$\Box$) we get the *UE implication system* for $\mathbb{D}_C^+ \cup \mathbb{E}^+$.

Note that k (and l) can be 0 for each of the rules except (E2) (irrelevant case). It will be shown that all relevant information regarding the constraints hold at a particular case can be deduced without applying (E3) for $k = 0$ (this case corresponds to $(*)$ in the NST system). This statement as well as the equivalence

[6]Note that $X \nrightarrow Z$ holds at least the same or even more information than δ.

between UE and the extended Armstrong implication system over $\mathbb{D}_C^+ \cup \mathbb{E}^+$ are formalized by the following lemmas. Notation $\vdash_A$ is used for provability using the Armstrong system for functional dependencies (the axiom and the rules (1) and (2) presented in section 2.3.3, over $\mathbb{D}$) and $\vdash_{EA}$ for provability using the extended Armstrong system (over $\mathbb{D} \cup \mathbb{E}$, extended with the rest of the rules). These systems are known to be sound and complete.

Lemma 3 (U) is sound, i.e. $\vdash_A$(U).

Lemma 4 System U is complete for functional dependencies with the restriction that trivial dependencies might not be proved and dependencies with more than one attribute on their right-hand sides are represented as canonical dependencies. More precisely, let $\mathcal{F}$ be a finite subset of $\mathbb{D}$. Then, for each two sets of attributes X and Z, $\mathcal{F} \vdash_A X \to Z \Longrightarrow \forall B \in Z \setminus X : \mathcal{F}' \vdash_U X \to B$ where $\mathcal{F}' = \{V \to C \mid \exists W : V \to W \in \mathcal{F}, C \in W \setminus V\}$.

Lemma 5 (E1), (E2), (E3) and (E□) are sound, i.e. $\vdash_{EA}$\{(E1), (E2), (E3)\} and if $\mathcal{F} \vdash_{UE} \neg$(E□) for a finite set $\mathcal{F} \subset \mathbb{D}_c^+ \cup \mathbb{E}^+$, then $\mathcal{F}$ is contradictory.

Lemma 6 System UE is complete for excluded functional constraints when the initial set of constraints is not contradictory and with the restriction that trivial constraints might not be proved. Positive constraints with more than one attribute on their right-hand sides are represented by canonical constraints (similarly to Lemma 4) and each non-self-contradictory trivial negated constraint is represented as a non-trivial negated constraint. More precisely,

- let $\mathcal{F}$ be a finite, non-contradictory subset of $\mathbb{D} \cup \mathbb{E}$ and let X and Z be two sets of attributes such that $Z \setminus X \neq \emptyset$. If $\mathcal{F} \vdash_{EA} X \nrightarrow Z$, then $\mathcal{F}' \vdash_{UE} X \nrightarrow Z \setminus X$, where $\mathcal{F}' = \{V \to C \mid \exists W : V \to W \in \mathcal{F}, C \in W \setminus V\} \cup \{V \nrightarrow U \mid \exists W : V \nrightarrow W \in \mathcal{F}, U = W \setminus V\}$.
- Additionally, if $\mathcal{F}' \vdash_{UE} X \nrightarrow V$ for a non-trivial negated constraint (and $\mathcal{F}'$ is not contradictory), then $\exists U \subseteq V$, $U \neq \emptyset$ such that $X \nrightarrow U$ can be deduced in the UE system without using (E3) for $k = 0$.
- If $\mathcal{F}$ is contradictory and the corresponding $\mathcal{F}' \subset \mathbb{D}_c^+ \cup \mathbb{E}^+$ (i.e. $\mathcal{F}$ contains no trivial excluded constraints and so $\mathcal{F}'$ contains no constraints in the form $X \nrightarrow \emptyset$), then $\mathcal{F}' \vdash_{UE} \neg$(E□), and deduction can be performed without using (E3) with $k = 0$.

Equivalence between systems ST and U, as well as between NST and UE is stated by the following lemmas:

Lemma 7 The implication systems ST and U are equivalent over $\mathbb{D}_c^+$, i.e. $\vdash_U$\{(S), (T)\} and $\vdash_{ST}$(U).

Lemma 8 The implication systems NST and UE are equivalent over $\mathbb{D}_c^+ \cup \mathbb{E}^+$, i.e. additionally to Lemma 7,

- $\vdash_{UE}$\{(NS), (NT1), (NT2), (∗)\},
- \{(S), (NT2)\}$\vdash_{NST}$(NT2S), $\neg$(E□)$\vdash_{NST}$ $\neg$(□),
- $\vdash_{NST}$\{(E1), (E2), (E3)\}, $\neg$(□)$\vdash_{UE}$ $\neg$(E□).
- Moreover, (∗) is only needed to deduce (E3) when $k = 0$ and vica versa (neither for deducing the other rules nor for deducing (E□) from (□) and vica versa).

7.5 Proofs of Lemmas and Theorems

Lemmas 3–8 are proved first, followed by 1 and 2. Proofs of Theorems 1, 4, 2 and 3 will follow afterwards, respectively. The order is based on dependencies between lemmas and theorems.

Proof of Lemma 3

For $k = 0$ the result is obvious by using (1) for $V = \emptyset$. Assume now that $k > 0$ and so $XY \to A_1, \cdots, XY \to A_k$ and $YA_1 \cdots A_k \to B$ hold. The axiom states $XY \to Y$. Applying rule (8) several times we get $XY \to YA_1 \cdots A_k$ and using rule (2) afterwards results $XY \to B$.

□

Proof of Lemma 4

Let $\mathcal{F} \vdash_A X \to Z$. We show $\mathcal{F}' \vdash_U \mathcal{G} = \{X \to B \mid B \in Z \setminus X\}$ by constructing a proof for that in system U, parallel with the proof of $X \to Z$ in the Armstrong system. Suppose the sequence $\langle f_1, \cdots, f_m\rangle$ is a proof of $X \to Z$. Construction of the new proof is made by ensuring for each item $f_i = V \to W$, elements of the set $\mathcal{F}'_i = \{V \to C \mid C \in W \setminus V\}$ already occur in the new proof before f_{i+1} is considered, i.e. for each i there exists a prefix $\langle f'_1, \cdots, f'_j\rangle$ in the new proof so that $\bigcup_{p=1}^{i} \mathcal{F}'_p = \bigcup_{q=1}^{j}\{f'_q\}$. For a particular i, this property is denoted by $\mathcal{P}_i$. Note that this set is a subset of $\mathbb{D}_c^+$ so the proof being constructed is valid in system U over $\mathbb{D}_c^+$ if we use rule (U) only. We start with the empty sequence as a prefix when no f_i's have been considered yet.

Let $0 < i \leq m$ and suppose by induction $\langle f'_1, \cdots, f'_j\rangle$ is the prefix of the new proof already constructed according to $\langle f_1, \cdots, f_i\rangle$ with the property just mentioned ($\mathcal{P}_i$). We perform an (optional) extension resulting $\langle f'_1, \cdots, f'_j, f'_{j+1}, ..., f'_{j+h}\rangle$ so that $\mathcal{F}'_{i+1} \subseteq \bigcup_{q=1}^{j+h}\{f'_q\}$, resulting that $\mathcal{P}_{i+1}$ holds. This will complete the proof since $\mathcal{F}'_m = \mathcal{G}$ because $f_m = X \to Z$ and m is reached by induction.

Let $f_{i+1} \in \mathcal{F}$. In this case, we simply add the elements of the set $\mathcal{F}'_{i+1}$ to the new proof. This step is valid since $\mathcal{F}'_i \subseteq \mathcal{F}'$.

If f_{i+1} is an instance of axiom $XY \to Y$ then $F'_{i+1} = \emptyset$ and nothing is to be done to ensure $\mathcal{P}_{i+1}$.

Let $f_{i+1} = X'V'W' \to Y'V'$ be a result of the application of rule (1) to a previous $f_{i'} = X' \to Y'$. This case, the set $\mathcal{F}'_{i+1} = \{X'V'W' \to D \mid D \in Y'V' \setminus X'V'W'\}$ must be added to the prefix of the new proof if not empty. For a particular element $X'V'W' \to D$ of $\mathcal{F}'_{i+1}$, $D \in Y'V' \setminus X'V'W' = Y' \setminus X'V'W' \subseteq Y' \setminus X'$ and so $X' \to D \in \mathcal{F}'_{i'}$, already occuring in the new proof. Applying rule (U) with $k = 0$, we get $X'V'W' \to D$ and we extend the new proof by this dependency. This step can be performed for each element of $\mathcal{F}'_{i+1}$, resulting $\mathcal{P}_{i+1}$ is true.

The remaining case is when $f_{i+1} = X' \to Z'$ is a result of the application of rule (2) to previous items $f_{i_1} = X' \to Y'$ and $f_{i_2} = Y' \to Z'$. If $\mathcal{F}'_{i+1} \neq \emptyset$, we have to include each $X' \to E \in \mathcal{F}'_{i+1}$ to the new proof by deducing it from previously added items. This will guarantee the property $\mathcal{P}_{i+1}$ holds. For such a dependency of $\mathcal{F}'_{i+1}$, $E \in Z' \setminus X'$. If $E \in (Z' \setminus C) \cap Y' \subseteq X' \setminus Y'$, then $X' \to E \in \mathcal{F}'_{i_1}$ and since it was already added to the proof when f_{i_1} was considered, nothing is to be done. Otherwise, $E \in Z' \setminus X'Y' \subseteq Z' \setminus Y'$ and $Y' \to E$ was already added to the proof when f_{i_2} was considered. Let $X'' = X' \setminus Y'$, $Y'' = Y' \cap X'$ and $\{A_s \mid 1 \leq s \leq k\} = Y' \setminus X'$ as $|Y' \setminus X'| = k$ ($k = 0$ is allowed, it means no A_s's exist). With these notations, $Y' \to E = Y''A_1 \cdots A_k \to E$ and $\forall s \in [1..k] : X' \to A_s = X''Y'' \to A_s \in \mathcal{F}'_{i_1}$. Since elements of $\mathcal{F}'_{i_1}$ already occur in the constructed prefix of the proof, applying (U) with X'' as X and Y'' as Y is a valid step and the resulting dependency $X''Y'' \to E = X' \to E$ can be added to the new proof.

□

Proof of Lemma 5

$V \rightarrow A_1 \cdots A_k$ can be deduced from $V \rightarrow A_1, \cdots, V \rightarrow A_k$ using (8) several times. We treat this as the first step for each case (with the corresponding V which can be XY, X or XZW).

We start with verifying (E1). Case $k = 0$ (and so $l = 0$) follows using the axiom and rule (3). Let $k > 0$, assume $XY \rightarrow A_1 \cdots A_k$, and $XY \not\twoheadrightarrow ZA_1 \cdots A_l$ for a $0 \leq l \leq k$. By using rule (1) we get $XY \rightarrow YA_1 \cdots A_k$, then $YA_1 \cdots A_k \not\twoheadrightarrow ZA_1 \cdots A_l$ follows according to rule (3). We finally get $YA_1 \cdots A_k \not\twoheadrightarrow Z$ by applying rule (5).

(E2) immediately follows from rule (6) (case $k = 0$ is irrelevant).

Now, for (E3), assume first that $k > 0$ and $XZW \rightarrow A_1 \cdots A_k$ and $XY \not\twoheadrightarrow ZA_1 \cdots A_k$ hold ($ZW \neq \emptyset$). We extend the first dependency using (1) to $XZW \rightarrow ZA_1 \cdots A_k$. Applying (7) then results $XY \not\twoheadrightarrow YZW$ and we get the desired constraint $XY \not\twoheadrightarrow ZW$ by rule (5). Case $k = 0$ corresponds to rule (4) which can be viewed as a special case of (7) as well ($Z \subseteq Y$).

Because the rules of system UE are sound, for each finite $\mathcal{F} \subset \mathbb{D}_c^+ \cup \mathbb{E}^+$, $\mathcal{F} \vdash_{UE} \neg(\text{E}\square) \Longrightarrow \mathcal{F} \vdash_{EA} \neg(\text{E}\square)$. By using rule (6) of the extended Armstrong system repeatedly, starting from $\neg(\text{E}\square)$, e.g. $X \rightarrow A_1$ and X $\not A_1$ can be derived, resulting that $\mathcal{F}$ is contradictory.

$\square$

Proof of Lemma 6

Let $\mathcal{F} \vdash_{EA} X \not\twoheadrightarrow Z$, $Z \setminus X \neq \emptyset$ and $\mathcal{F}$ is not contradictory. Note that $\mathcal{F}'$ is not contradictory as well, since it is equivalent to $\mathcal{F}$ in terms of the extended Armstrong system. We show that for at least one element f'' of the set $\mathcal{H} = \{X \rightarrow U \mid U \subseteq Z \setminus X, U \neq \emptyset\}$, $\mathcal{F}' \vdash_{UE} f''$ holds by constructing a proof for it in system UE without using rule (E3) as $k = 0$. This verifies the second statement. Completeness then simply follows by using (E3) as $k = 0$ as one more step at the end of the proof, resulting $\mathcal{F}' \vdash_{UE} X \not\twoheadrightarrow Z$.

Similarly to Lemma 4, we construct the new proof for an element of $\mathcal{H}$ in the system UE, parallel with the proof of $X \not\twoheadrightarrow Z$ in the extended Armstrong system. Suppose the sequence $\langle f_1, \cdots, f_m \rangle$ is a proof of $X \not\twoheadrightarrow Z$. Each item f_i can be either a functional dependency or an excluded constraint. For functional dependencies, the same method can be used as discussed in the proof of Lemma 4 so that elements of the set $\mathcal{F}'_i = \{V \rightarrow C \mid C \in W \setminus V\}$ already occur in the new proof before f_{i+1} is considered. For each excluded constraint $f_i = V \not\twoheadrightarrow W$, $W \setminus V \neq \emptyset$ (since $\mathcal{F}$ is not contradictory[7]), therefore the set $\mathcal{F}''_i = \{V \rightarrow U \mid U \subseteq W \setminus V, U \neq \emptyset\}$ is nonempty. We ensure at least one element of $\mathcal{F}''_i$ occurs in the new proof (note that $\mathcal{F}''_i \neq \emptyset$) before f_{i+1} is considered. This property can be formalized by defining that $\mathcal{F}''_i = \emptyset$ for functional dependencies and $\mathcal{F}'_i = \emptyset$ for excluded constraints: for each i there exists a prefix $\langle f'_1, \cdots, f'_j \rangle$ in the new proof so that $\bigcup_{p=1}^{i} \mathcal{F}'_p \subseteq \bigcup_{q=1}^{j} \{f'_q\}$, $\bigcup_{p=1}^{i} (\mathcal{F}'_p \cup \mathcal{F}''_p) \supseteq \bigcup_{q=1}^{j} \{f'_q\}$ and $\forall p \in [1..i] : F''_p \neq \emptyset \Longrightarrow \mathcal{F}''_p \cap \bigcup_{q=1}^{j} \{f'_q\} \neq \emptyset$. Denote this property by $\mathcal{P}_i$. Since $\bigcup_{q=1}^{j} \{f'_q\} \subset \mathbb{D}_c^+ \cup \mathbb{E}^+$ holds, the proof to be constructed is valid in system UE over $\mathbb{D}_c^+ \cup \mathbb{E}^+$ if we use rules of system UE only. We start with the empty sequence as a prefix when no f_i's have been considered yet.

Let $0 < i \leq m$ and suppose by induction that $\langle f'_1, \cdots, f'_j \rangle$ is the prefix of the new proof already constructed according to $\langle f_1, \cdots, f_i \rangle$ with the property $\mathcal{P}_i$. Case $f_{i+1} \in \mathbb{D}$ (a positive element of $\mathcal{F}$, an instance of the axiom or a result of rule (1) or (2)) is already discussed in the proof of Lemma 4. If $f_{i+1} \in \mathbb{E}$, then $\mathcal{F}''_{i+1} \neq \emptyset$, we perform an (optional) extension resulting $\langle f'_1, \cdots, f'_j, f'_{j+1}, ..., f'_{j+h} \rangle$ so that $\mathcal{F}''_{i+1} \cap \bigcup_{q=1}^{j+h} \{f'_q\} \neq \emptyset$. This will complete the proof since $f_m = X \not\twoheadrightarrow Z$, $\mathcal{F}''_m = \mathcal{H}$ and m is reached by induction.

[7]A self-contradictory constraint (a negated constraint of the form $XY \not\twoheadrightarrow X$) can be deduced using the extended Armstrong implication system if and only if the initial set of constraints is contradictory. The 'only if' direction is trivial. To see the 'if' direction, apply rule (3) for the case $Y = Z$ or rule (6) for $Y = \emptyset$ or rule (7) for $X = Y$.

Let $f_{i+1} \in \mathcal{F} \cap \mathbb{E}$, $f_i = X' \nrightarrow Z'$. In this case, $X' \nrightarrow Z' \setminus X' \in F''_i$ and we add this constraint to the new proof to make sure $\mathcal{P}_{i+1}$ holds.

If $f_{i+1} = Y' \nrightarrow Z' \, (Z' \setminus Y' \neq \emptyset)$ is derived by applying rule (3) to previous items $f_{i_1} = X' \rightarrow Y'$ and $f_{i_2} = X' \nrightarrow Z' \, (Z' \setminus X' \neq \emptyset)$. We have to extend the new proof with a $Y' \nrightarrow V'$ for a nonempty $V' \subseteq Z' \setminus Y'$. Let $X' \nrightarrow U' \in F''_{i_2} \, (U' \subseteq Z' \setminus X')$ that already occurs in the constructed prefix of the proof. Let $X'' = X' \setminus Y'$, $Y'' = X' \cap Y'$, $Z'' = U' \setminus Y' \subseteq Z' \setminus Y'$, $\{A_1, \cdots, A_k\} = Y' \setminus X'$ as $|Y' \setminus X'| = k$ ($k = 0$ is allowed resulting no $A'_s s$) and suppose $A_1, \cdots, A_l$ are the elements of $U' \cap Y'$, a subset of $Y' \setminus X'$. This way, $Z''A_1 \cdots A_l = U'$ and $X''Y'' \nrightarrow Z''A_1 \cdots A_l = X' \nrightarrow U'$. For each $s \in [1..k]$, $X' \rightarrow A_s = X''Y'' \rightarrow A_s \in \mathcal{F}'_{i_1}$ and therefore, they were added to the new proof when f_{i_1} was considered. Applying rule (E1) with X'' as X, Y'' as Y and Z'' as Z, we get $Y''A_1 \cdots A_k \nrightarrow Z'' = Y' \nrightarrow Z''$ if $Z'' \neq \emptyset$. The proof being constructed is extended by this negated constraint, and $Z'' \subseteq Z' \setminus Y'$ ensures the desired property $\mathcal{P}_{i+1}$ holds. We show that Z'' cannot be empty. Suppose it is. Then, $X' \nrightarrow U' = X' \nrightarrow A_1 \cdots A_l$. Since each $X' \nrightarrow A_s$ was derived in system UE, we would get $\neg(\mathrm{E}\Box)$ which means F' is contradictory (see Lemma 5, the system UE is sound). But this contradicts our condition that F' is not contradictory, hence, $Z'' \neq \emptyset$.

If $f_{i+1} = X' \nrightarrow Y'Z' \, (Y'Z' \setminus X' \neq \emptyset)$ is a result of applying rule (4) to a previous constraint $f_{i'} = X' \nrightarrow Y'$, then by induction, a constraint $X' \nrightarrow U' \in F''_{i'} \, (U' \subseteq Y' \setminus X', U' \neq \emptyset)$ must already exist in the constructed prefix of the new proof. Since $U' \subseteq Y' \setminus X' \subseteq Y'Z' \setminus X'$, $F''_{i'} \subseteq F''_{i+1}$ and the property $\mathcal{P}_{i+1}$ holds without performing an extension.

Consider an $f_{i+1} = X'Z' \nrightarrow Y' \, (Y' \setminus X'Z' \neq \emptyset)$ as the next case, derived from a previous constraint $f_{i'} = X'Y' \nrightarrow Y'Z'$ of the original proof using rule (5). By induction, a constraint $X'Z' \nrightarrow U' \in F''_{i'} \, (U' \subseteq Y'Z' \setminus X'Z', U' \neq \emptyset)$ must exist in the prefix already constructed. Again, no extension is necessary for $\mathcal{P}_{i+1}$ to be true, since $Y'Z' \setminus X'Z' = Y' \setminus X'Z'$ and so $F''_{i'} = F''_{i+1}$.

Suppose now that $f_{i+1} = X' \nrightarrow Y' \, (Y' \setminus X' \neq \emptyset)$ and is derived applying rule (6) to previous items $f_{i_1} = X' \rightarrow Z'$ and $f_{i_2} = X' \quad \not Y'Z'$. To ensure $\mathcal{P}_{i+1}$, we need to provide a constraint $X' \nrightarrow V' \in F''_{i+1} \, (V' \subseteq Y' \setminus X', V' \neq \emptyset)$ will be included into the new proof. We are supposing by induction that $X' \nrightarrow U' \in F''_{i_2}$ is already included for a nonempty set $U' \subseteq Y'Z' \setminus X'$ and we have nothing to do if $U' \subseteq Y' \setminus X'$. Let $\{A_1, \cdots A_k\} = (Z' \setminus X') \cap U'$ with the A_s's different ($k = 0$ is allowed), $Z'' = U' \setminus Z'$. Hence, $X' \nrightarrow U' = X' \nrightarrow Z''A_1 \cdots A_k$. Because of each $X' \rightarrow A_s \in F'_{i_1}$, they are already deduced and exist in the new proof. $Z'' \neq \emptyset$ can be shown similarly to the case of rule (3) above, therefore, rule (E2) can be applied with Z'' as Z and the result $X' \nrightarrow Z'' = X' \nrightarrow U' \setminus Z' \, (U' \setminus Z' \subseteq Y' \setminus X')$ can be added to the new proof ensuring $\mathcal{P}_{i+1}$.

The last case is when $f_{i+1} = X' \nrightarrow Y' \, (Y' \setminus X' \neq \emptyset)$ is a result of using rule (7) starting with previously derived constraints $f_{i_1} = Y' \rightarrow Z'$ and $f_{i_2} = X' \nrightarrow Z' \, (Z' \setminus X' \neq \emptyset)$. Providing a $X' \nrightarrow V' \in \mathcal{F}''_{i+1} \, (V' \subseteq Y' \setminus X', V' \neq \emptyset)$ exists in the new proof is necessary for $\mathcal{P}_{i+1}$ to be true. By induction, an element $X' \nrightarrow U' \in F''_{i_2} \, (U' \subseteq Z' \setminus X', U' \neq \emptyset)$ is already deduced and included into the new proof being constructed. If $U' \setminus Y' = \emptyset$, then $U' \subseteq Y' \setminus X'$ and P_{i_2} implies P_{i+1}. Otherwise, let $\{A_1, \cdots A_k\} = U' \setminus Y'$ as $|U' \setminus Y'| = k \, (k > 0)$ and $Z'' = U' \cap Y'$, $Y'' = X' \cap Y'$, $X'' = X' \setminus Y'$, $W'' = Y' \setminus X'U'$. With these notations, $U' = Z''A_1 \cdots A_k$, $X' = X'' \setminus Y''$ and $Y' = Y''Z''W''$. For each $s \in [1..k]$, $Y' \rightarrow A_s = Y''Z''W'' \rightarrow A_s \in F'_{i_1}$ and they were added to the new proof when f_{i_1} was considered. Applying (E3) (with $k > 0$, X'' as X, Y'' as Y, Z'' as Z and W'' as W) to these together with $X' \nrightarrow U' = X''Y'' \nrightarrow Z''A_1 \cdots A_k$, the result is $X''Y'' \quad \not Z''W''$. Note that $Y'' \cap Z'' = X' \cap Y' \cap U'$ which is empty since $U' \subseteq Z' \setminus X'$. Hence, $U' \cap Y' \subseteq Y' \setminus X'$ and $Z''W'' = (U' \cap Y') \cup (Y' \setminus X'U') = Y' \setminus X' \neq \emptyset$. Therefore, adding the deduced constraint to the new proof ensures $\mathcal{P}_{i+1}$ which completes our construction.

Construction of the proof for an element of $\mathcal{H}$ in the UE system can be performed by carrying over the above process for each element f_{i+1} of the original proof of $X \nrightarrow Z$ in the extended Armstrong system. Note we did not use the rule (E3) with $k = 0$ for any of the cases, verifying the second statement of the lemma.

Let us consider the final statement of this lemma regarding to the case when $\mathcal{F}$ is contradictory without containing trivial negated constraints. Suppose $\mathcal{F} \vdash_{EA} X \rightarrow Y$ for sets of attributes X and Y and $\mathcal{F} \vdash_{EA} X \nrightarrow Y$ at the same time. Let $\{C_1, \cdots, C_k\} = Y \setminus X$. Since rules (3)-(7) can not be used to deduce positive constraints, $\mathcal{F} \vdash_A X \rightarrow Y$ must hold which implies $\mathcal{F}' \vdash_U \{X \rightarrow C_i\}$ for each $i \in [1..k]$ according to Lemma 4. We show that either $\mathcal{F}' \vdash_{UE} X \nrightarrow C_1 \cdots C_l$ for an $l \in [1..k]$, or $\neg(\text{E}\Box)$ (with possibly other attributes) can be deduced instead. Let us start the construction of the proof of $X \nrightarrow C_1 \cdots C_k$ in system UE, parallel to the original proof of $X \nrightarrow Y$ the same way as discussed for the non-contradictory case. Reviewing the method, the only case of failure is when a self-contradictory constraint $X'Y' \nrightarrow X'$ arises in the original proof. If we let the empty set as being the right-hand side of negated constraints in UE for a while, the construction could be carried over, resulting that $X'Y' \nrightarrow \emptyset$ should be included to the new proof by using one of the rules (E1), (E2) or (E3). Instead of performing this step, it can easily be realized that the precondition for this step ($Z = \emptyset$ for (E1) or (E2) and $ZW = \emptyset$ for (E3)) is exactly $\neg(\text{E}\Box)$ with some attributes $A_1, \cdots, A_{k'}$. We conclude that $\neg(\text{E}\Box)$ must be already deduced if construction fails, otherwise, $X \nrightarrow C_1 \cdots C_k$ can be deduced for an $l \in [1..k]$ which corresponds to $\neg(\text{E}\Box)$ as well. Note we still did not have to use (E3) with $k = 0$ for deducing $\neg(\text{E}\Box)$.

$\Box$

Proof of Lemma 7

Rules (S) and (T) are special cases of (U) with $k = 0$, $|X| = 1$ and $k = 1$, $|X| = 0$, respectively.

Let $k > 0$ and $|X| > 0$. We show that for each $0 \leq t \leq k$, $\{XY \rightarrow A_1, \cdots, XY \rightarrow A_k, YA_1 \cdots A_k \rightarrow B\} \vdash_{ST} XYA_1 \cdots A_t \rightarrow B$ hold ($t = 0$ corresponds our goal). Suppose, by induction for t that $XYA_1 \cdots A_{t+1} \rightarrow B$ is deducible. $XY \rightarrow A_{t+1}$ can be extended to get $XYA_1 \cdots A_t \rightarrow A_{t+1}$ using (S). Now we can immediately deduce $(XYA_1 \cdots A_t \rightarrow B)$ by using (T). Case $t = 0$ follows by induction.

$\Box$

Proof of Lemma 8

Equivalence of $\{(\text{U})\}$ and $\{(\text{S}), (\text{T})\}$ was stated and proven by Lemma 7. We need to consider the rest of the rules dealing with negated dependencies. (NS) and (NT1) are special cases of (E1) with $|X| = 1$, $k = 0$ ($l = 0$) and $|X| = 0$, $k = 1$, $l = 0$, respectively. Similarly, (NT2) and $(*)$ are special cases of (E3) with $k = 1$ and $k = 0$, respectively.

(NT2S) can be easily simulated by extending $Y \rightarrow B$ to $YZ \rightarrow B$ using (S) and using (NT2) with $W = \emptyset$ to get $Y \nrightarrow Z$ afterwards.

Starting from $\neg(\text{E}\Box)$, $\neg(\Box)$ can be derived by the repeated use of rule (NT2S). The reversed case is trivial since $(\Box)$ is a special case of $(\text{E}\Box)$ with $k = 1$.

Consider (E1) with $k = 0$, $l = 0$, $|X| > 1$. This case can easily be simulated by repeatedly using (NS). Let $k > 0$. If $l > 0$, (NT2S) can be applied to remove A_i's ($1 \leq i \leq l$) step-by-step from the right-hand-side of $XY \nrightarrow ZA_1 \cdots A_l$ (dependency $XY \rightarrow A_i$ holds for each required i). The remaining case for (E1) is $k > 0$, $l = 0$. Applying (NT1) results $XYA_1 \nrightarrow Z$. We extend each dependency $XY \rightarrow A_j$ ($1 < j \leq k$) to $XYA_1 \cdots A_{j-1} \rightarrow A_j$ using (S) to allow (NT1) be applied repeatedly. This way we get $XYA_1 \cdots A_j \nrightarrow Z$ for each j. As $j = k$ reached, we get the desired result $YA_1 \cdots A_k \nrightarrow Z$ by using rule (NS) to remove X from the left-hand side.

In the case of (E2), only $k > 0$ is relevant and $X \nrightarrow ZA_1 \cdots A_j$ can be deduced for each $j \in [0..k]$ step-by-step using (NT2S). $j = 0$ corresponds to our goal $X \nrightarrow Z$.

Consider (E3) with $k > 0$. We extend $XZW \rightarrow A_k$ to $XZA_1 \cdots A_{k-1}W \rightarrow A_k$ using rule (S). Application of (N2) now results $XY \nrightarrow ZA_1 \cdots A_{k-1}W$ (attribute set $ZA_1 \cdots A_{k-1}$ corresponds to Z in the original formula of (N2)). Attributes $A_1 \cdots A_{k-1}$ of the right-hand side can be eliminated using (NT2S) repeatedly, resulting $XY \not\twoheadrightarrow ZW$ as desired.

The remaining case for (E3) is $k = 0$ which can be simulated by the (sometimes repeated) use of $(*)$. Note we did not need rule $(*)$ for any other case. This verifies the last statement of the lemma, together with the fact that each rule (and the axiom $(\square)$) of NST corresponds to a special case of a rule (or axiom $(\mathrm{E}\square)$) of UE, and $(*)$ is the only one for (E3) with $k = 0$.

$\square$

Proof of Lemma 1

Trivial: Rules (P) and (Q) are special cases of (NS) and (NT1), respectively ($|Z| = 1$, $B = Z$). Rule (R) is a special case of rule (NT2) ($Z = \emptyset$, $|W| = 1$, $A = W$).

$\square$

Proof of Lemma 2

Assume first that $\mathcal{F}$ is not contradictory (according to lemmas 6 and 8, this is equivalent with $\mathcal{F} \nvdash_{NST'} \neg(\square)$) and $\mathcal{F} \vdash_{NST'} X \not\twoheadrightarrow A$ holds. Consider a proof for the constraint $X \not\twoheadrightarrow A$. It clearly follows from the form of rules that the items of the proof can be reordered so that all functional dependencies precede the negated constraints. If we skip the items not needed for the deduction of $X \not\twoheadrightarrow A$, we notice that only the first negated constraint used in the proof is needed to be taken from $\mathcal{F}$ since there is no rule with two or more negated constraint as preconditions. Moreover, the rest of the negated constraints can be ordered so that each of them is derived applying a rule with the previous negated constraint (together with a positive constraint for some rules). Denote by $\langle \delta_1, \cdots, \delta_s, \varepsilon_t, \cdots, \varepsilon_1 \rangle$ this reduced, reordered proof using the NST' system ($\delta_i \in \mathbb{D}_c^+$, $\varepsilon_j \in \mathbb{E}_c^+$ for each $i \in [1..s]$ and $j \in [1..t]$). Note that $\varepsilon_1 = X \not\twoheadrightarrow A$ and since $\varepsilon_t \in \mathcal{F}$ and it is the only item ε_i with this property, $(\mathcal{F} \cap \mathbb{D}_c^+) \cup \{\varepsilon_t\} \vdash_{NST'} X \not\twoheadrightarrow A$ holds. We construct an equivalent proof using the system PQRST parallel to the original one. The positive part of the proof ($\langle \delta_1, \cdots, \delta_s \rangle$) is kept as the prefix since rules (S) and (T) are common for the two systems NST' and PQRST, the negative part ($\langle \varepsilon_t, \cdots, \varepsilon_1 \rangle$) can be replaced by another sequence $\langle \delta_{s+1}, \cdots \delta_r, \zeta_u, \cdots, \zeta_1 \rangle$ ($r \geq s$, $u \in \mathbb{N}^+$; $\delta_i \in \mathbb{D}_c^+$, $\zeta_j \in \mathbb{E}_c^+$ for each $i \in [s+1..r]$ and $j \in [1..u]$), resulting a valid proof using the PQRST system over $\mathbb{D}_c^+ \cup \mathbb{E}_c^+$. Construction is made by ensuring that for each $i \in [1..t]$ there exists a prefix $\langle \delta_1, \cdots, \delta_{h_i} \rangle$ and a postfix $\langle \zeta_{j_i}, \cdots, \zeta_1 \rangle$ of the new proof ($h_i \geq h_{i-1} \geq s$ and $j_i \geq j_{i-1}$ for each $i \in [2..t]$) so that $\langle \delta_1, \cdots, \delta_{h_i}, \zeta_{j_i}, \cdots, \zeta_1 \rangle$ is a valid proof for $(\mathcal{F} \cap \mathbb{D}_c^+) \cup \{\varepsilon_i\} \vdash_{PQRST} X \not\twoheadrightarrow A$. Furthermore, it is ensured that $\zeta_{j_i} = V \not\twoheadrightarrow A$ for each $\varepsilon_i = V \not\twoheadrightarrow W$ and $VA \to E \in \{\delta_1, \cdots, \delta_{h_i}\}$ for each $E \in |W \setminus A|$. Denote these properties by $\mathcal{Q}_i$ for a particular $i \in [1..t]$. $\mathcal{Q}_1$ holds by choosing ζ_1 as $\zeta_1 = \varepsilon_1$, $h_1 = s$ and $j_1 = 1$. t will be reached by induction, following the steps described below. If $\zeta_{j_t} = \varepsilon_t$, then $u = j_t$ and the proof is complete. Otherwise, $\zeta_{j_t} = Y \not\twoheadrightarrow A$, $\varepsilon_t = Y \not\twoheadrightarrow E\,(A \neq E)$ and according to $\mathcal{Q}_t$, $YA \to E$ appears in the prefix already constructed. Adding $\zeta_{j_t+1} = \varepsilon_t$ as the first negated constraint to the postfix completes the proof (let $u = j_t + 1$) since ζ_{j_t} can be deduced applying rule (R) with $YA \to E$ and ε_t. This is the only necessary application of rule (R) as it will be shown.

Assume by induction that a prefix $\langle \delta_1, \cdots, \delta_{h_i} \rangle$ and a postfix $\langle \zeta_{j_i}, \cdots, \zeta_1 \rangle$ of the new proof using the PQRST system is already constructed considering the postfix $\langle \varepsilon_i, \cdots, \varepsilon_1 \rangle$ of the original proof ($i \in [1..t-1]$), and $\mathcal{Q}_i$ holds. We optionally perform an extension on the prefix or the postfix (or both), ensuring the property $\mathcal{Q}_{i+1}$. Several subcases arise, depending on which rule was used to derive ε_i from ε_{i+1} ($\varepsilon_i \notin \mathcal{F}$ as seen above).

Assume $\varepsilon_i = Y \not\twoheadrightarrow Z$ and $\varepsilon_{i+1} = YC \not\twoheadrightarrow Z$ (rule (NS) was used in the original proof) and $\zeta_{j_i} = Y \not\twoheadrightarrow A$. Let $j_{i+1} = j_i + 1$, $\zeta_{j_{i+1}} = YC \not\twoheadrightarrow A$ (it is a valid step since ζ_{j_i} can be derived applying rule (P) with $\zeta_{j_{i+1}}$). To satisfy $\mathcal{Q}_{i+1}$, we extend the prefix already constructed with each of the dependencies $YCA \to E\,(E \in Z \setminus A)$

not occuring in the prefix yet. Such a dependency can be deduced by applying rule (S) on $YA \to E$ which already exists in the prefix by induction. Note that $A \neq C$ since $\mathcal{F}$ is not contradictory ($A = C$ would mean $YC \to E$ for each $E \in Z \setminus C = Z$ and this would contradict $YC \not\twoheadrightarrow Z$).

The second case is when rule (NT1) was applied on $\varepsilon_{i+1} = Y \not\twoheadrightarrow Z$ and a $\delta_k = Y \to A'$ ($k \leq s$) to get $\varepsilon_i = YA' \not\twoheadrightarrow Z$. By induction, $\zeta_{j_i} = YA' \not\twoheadrightarrow A$ and $\forall E \in Z \setminus A\, \exists l \in [1..h_i] : \delta_l = YA'A \to E$. The postfix can be extended by $\zeta_{j_{i+1}} = \zeta_{j_i+1} = Y \not\twoheadrightarrow A$ since ζ_{j_i} can be deduced using rule (Q) in one step, starting with δ_k and this $\zeta_{j_{i+1}}$. Furthermore, $YA \to E$ can be deduced for each $E \in Z \setminus A$ using rule (S) and (T), starting with the already existing $Y \to A'$ and $YA'A \to E$. We extend the prefix with items $YA \to A'$ and $YA \to E$ if they do not occur in the prefix yet, ensuring $\mathcal{Q}_{i+1}$.

The third and final case is of rule (NT2). Let $\varepsilon_i = Y \not\twoheadrightarrow ZW$, $\varepsilon_{i+1} = Y \not\twoheadrightarrow ZB$ and k such that $\delta_k = YZW \to B$ ($k \leq s$). By induction, $\zeta_{j_i} = Y \not\twoheadrightarrow A$ and $\forall E \in ZW \setminus A\, \exists l \in [1..h_i] : \delta_l = YA \to E$. Let $j_{i+1} = j_i$ (no constraint is added to the postfix). Ensuring $YA \to E$ occurs in the prefix of the new proof for each $E \in ZB \setminus A$ provides Q_{i+1}. Since it occurs for each $E \in Z \setminus A$, only one dependency $YA \to B$ is needed to be included into the new proof if it does not occur yet (in this case, $A \neq B$). Rule (U) can be used to derive $YA \to B$ from δ_k and δ_l's mentioned above. According to Lemma 7, rule (U) can be simulated with rules (S) and (T). We extend the prefix of the new proof with steps of this deduction.

Construction of the new proof using system PQRST can be performed for a non-contradictory set by carrying over the above steps. If the initial set $\mathcal{F}$ is contradictory, then $\neg(\Box)$ can be deduced using system NST'. For such a proof, the above transformation process may fail if a $\varepsilon_k = Y \not\twoheadrightarrow Z$ exists such that $k > 1$ and $\mathcal{F} \vdash_{ST} Y \to E$ for each $E \in Z$ (see the case of rule (NS)). Selecting the first ε_k of the original proof (with the largest k possible) with this property and truncating the original proof at ε_k allows the transformation process to be carried out. We start by selecting an attribute $D \in Z$ arbitrarily for $\zeta_1 = Y \not\twoheadrightarrow D$ and adding the steps of deduction of $Y \to E$ for each $E \in Z$ to the prefix $\langle \delta_1, \cdots, \delta_s \rangle$ of the new proof and go on as discussed. The process will not fail and both $Y \to D$ and $Y \not\twoheadrightarrow D$ are deduced, indicating that $\mathcal{F}$ is contradictory.

□

Proof of Theorem 1

Soundness and completeness of ST follows using the soundness and completeness of the Armstrong system and lemmas 3, 4 and 7. Note that in Lemma 4, if $\mathcal{F} \in \mathbb{D}_c^+$ and $X \to Z \in \mathbb{D}_c^+$ as well, then $\mathcal{F} = \mathcal{F}'$ and there is exactly one $B \in Z \setminus X$ and $X \to Z = X \to B$ for this B. Therefore, we simply get $\mathcal{F} \vdash_{ST} \delta \iff \mathcal{F} \vdash_U \delta \iff \mathcal{F} \vdash_A \delta \iff \mathcal{F} \models \delta$.

□

Proof of Theorem 4

Part 1. Soundness and completeness of NST (without $(\Box)$) follows using the soundness and completeness of the Armstrong system and lemmas 5, 6 and 8. Note that in Lemma 6, if $\mathcal{F} \subset \mathbb{D}_c^+ \cup \mathbb{E}^+$ and $X \not\twoheadrightarrow Z \in \mathbb{E}^+$, then $\mathcal{F} = \mathcal{F}'$ and $X \not\twoheadrightarrow Z = X \not\twoheadrightarrow Z \setminus X$ with $Z \setminus X \neq \emptyset$. Similarly to Theorem 1, we get $\mathcal{F} \vdash_{NST} \delta \iff \mathcal{F} \vdash_{UE} \delta \iff \mathcal{F} \vdash_{EA} \delta \iff \mathcal{F} \models \delta$ for a non-contradictory set $\mathcal{F}$. Soundness holds regardless of $\mathcal{F}$ being contradictory or not, therefore, $\mathcal{F} \vdash_{UE}(\Box) \Longrightarrow \mathcal{F} \vdash_{EA}(\Box)$. The reverse direction for contradictory cases follows by Lemma 6, since $F = F' \subset \mathbb{D}_c^+ \cup \mathbb{E}^+$.

Part 2. Soundness over $\mathbb{D}_c^+ \cup \mathbb{E}^+$ and completeness over $\mathbb{D}_c^+$ is a trivial consequence of Part 1. Similarly to Part 1, $\mathcal{F}' = \mathcal{F}$. Using Lemmas 6 and 8 (focusing on their last statements), we get the following if $\mathcal{F}$ is not contradictory: $\mathcal{F} \Vdash X \not\twoheadrightarrow Y \in \mathbb{E}^+ \Longrightarrow F \vdash_{UE} X \not\twoheadrightarrow Y \Longrightarrow [\exists Z \subseteq Y, Z \neq \emptyset : \mathcal{F} \vdash_{UE} X \not\twoheadrightarrow Z$ without using (E3) for $k = 0] \Longrightarrow [\mathcal{F} \vdash_{NST} X \not\twoheadrightarrow Z$ for the same Z without using $(*)]$. For a contradictory set $\mathcal{F}$, we get $[\mathcal{F} \vdash_{UE} \neg(\mathrm{E}\Box)$ without using (E3) for $k = 0] \Longrightarrow [\mathcal{F} \vdash_{NST} \neg(\Box)$ without using $(*)]$.

□

Proof of Theorem 2

The statement immediately follows from Theorem 4 and Lemmas 1 and 2.

□

Proof of Theorem 3

Part 1. We show that applying (S) after (T) has been applied can be substituted by applying (T) twice after two subsequent application of (S). Suppose $YA \to B$ and $Y \to A$ hold. Applying (T) and then (S) results $Y \to B$ and $YC \to B$. Using (S) with both of the initial dependencies instead results $YCA \to B$ and $YC \to A$. By applying (T) with these two dependencies we get $YC \to B$. Hence, $Y \to B$ can be deduced now by applying (T) with the two initial dependencies.

Part 2. According to Part 1, all positive consequences can be deduced using (S) as many times as possible and then using (T) as many times as possible. It is clear that each application of (P), (Q) and (R) can be performed afterwards. We need to prove that an application of rule (R) after an application of rule (P) or (Q) can be substituted by another sequence so that (R) precedes all application of (P) and (Q).

As the first case, assume rule (R) is used directly after rule (Q). For three constraints $Y \to A$, $Y \not\to B$ and $YAC \to B$, we get $YA \not\to B$ using (Q) and then $YA \not\to C$ using (R). For constructing an alternative way of deriving $YA \not\to C$, note that $YC \to B$ must have been already deduced since it is a positive constraint that can be derived using rule (T). Applying rule (R) with $YC \to B$ and $Y \not\to B$ first, we get $Y \not\to C$ and the desired constraint $YA \not\to C$ follows using rule (Q). A second application of (Q) the same way as in the original version results $YA \not\to B$.

The second case is when rule (R) is used after rule (P): starting with constraints $YC \not\to B$ and $YA \to B$, the constraint $Y \not\to B$ follows using rule (P) and then $Y \not\to A$ is derived using rule (R). Note that $YAC \to B$ must have been already deduced using rule (S). Starting with this constraint together with $YC \not\to B$, constraint $YC \not\to A$ follows using rule (R), and $Y \not\to A$ can be derived using (P). A second application of (P) the same way as in the original version can be performed, resulting $Y \not\to B$.

We conclude that (R) can precede any application of (P) and (Q) without any loss of deduction power. It is trivial that the order of (P) and (Q) is arbitrary.

If $|\mathcal{G}| = 1$ then the proof can be transformed the same way as in the proof of Lemma 2 since rules of system PQRST are special cases of those in system NST' (see Lemma 1). Rule (R) is used only once in the transformed proof.

□

Chapter 8

Spreadsheet Representation & the Number of Constraint Sets

8.1 The Spreadsheet Notation of Sets of Functional Dependencies

A set of functional dependencies over a specific set of attributes can be represented as a row of a table where columns correspond to the possible functional dependencies and a digit 1 or 0 in a column indicates the presence or absence of the dependency. This representation is a brief but still convenient way to present a larger amount of sets and can also be used for reasoning on a particular set of constraints. Section 8.3 will show how implication of functional constraints can be performed using the spreadsheet representation.

For a binary relation over the attributes A and B, the possible functional dependencies in $\mathbb{D}_c^+$ are $B \to A$, $A \to B$, $\emptyset \to B$ and $\emptyset \to A$. The first two are one-dimensional, while the rest is zero-dimensional. The number of possible (closed) sets is 7 when the roles of A and B are different.

Since the main focus is on schema design, cases with zero-dimensional constraints (specifying constant attributes) are ignored. Two sets of constraints are equivalent if a permutation of attributes exists that transforms one set to another Treating equivalent sets as one single case we get the 3 possible binary cases presented in Section 2.2. Table 8.1 gives a simple example for the spreadsheet representation, extended with the conventional notation of functional dependencies and indicating the dimensions of attributes.

Similar representation will be used for ternary and quaternary relation schemata. Dependencies are grouped according to their dimensions. The possible dimension values for n attributes are $\{1, 2, \ldots, n-1\}$. The number of possible (singleton, nontrivial) dependencies with dimension d is $\binom{n}{d}(n-d)$. This way the

Case	FD's		Conventional	Dimension	
#	$B \to A$	$A \to B$	notation	$[A]$	$[B]$
# 0	0	0	$\emptyset$	∞	∞
# 1	1	0	$\{B \to A\}$	1	∞
# 2	1	1	$\{B \to A,\ A \to B\}$	1	1

Table 8.1: Spreadsheet representation of sets of functional dependencies for binary relations without constant attributes (sets equivalent up to attribute permutations treated as one case)

number of columns of the spreadsheet for n-ary cases becomes $\sum_{d=1}^{n-1} \binom{n}{d} (n-d)$. It equals 2 for 2 attributes, 9 for 3 attributes, 28 for 4 attributes and 75 for 5 attributes.

A representant set is selected for each case of equivalent sets by fixing the order of attributes. Representant sets of cases are ordered and grouped according to the dimensions of attributes. Only those groups (classes) are considered where dimensions of attributes form an increasing sequence ($[A] \leq [B] \leq [C] \leq \cdots$) since we do not get different cases (relationship types) by permuting the attributes.

8.2 Different Sets of Functional Constraints

8.2.1 Generating and Counting the Number of Sets

Generating and counting the number of (closed) sets for small relation schemata (with 1, 2, 3, 4 and 5 attributes) was performed for both all sets (where a set with its attributes permuted is considered as a different set) and different cases or relationship types (a permutation of attributes does not change the case). Moreover, both calculations were carried out for the cases with and without zero-dimensional constraints (constant attributes) as well. A summarizing table can be found in Section 8.2.4.

Computation was carried out by a back-tracking algorithm using the ST implication system (rules (S) and (T), see section 7.3). The algorithm works by systematically generating each possible closed set of dependencies. A set is generated by selecting some of the possible one-dimensional dependencies first. A step to one dimension higher is performed by generating dependencies using the extension rule (S) and the remaining possible dependencies of the same dimension give chance for selecting some of them into the set. This step is repeated until the maximal finite dimension is reached ($n-1$ if the number of attributes is n) or is noticed that the other rule (T) can be applied. In the latter case rule (T) is not applied because it would generate a dependency with one dimension lower but the selection for lower-dimensional dependencies is already fixed. Such a set is dropped and another selection is made. Since each possible selection is generated, rule (T) is not needed to be applied. The program is written in PROLOG since the logical approach this language provides is an obvious way of formalizing our problem and PROLOG supports backtracking in a natural way.

The refined algorithm works by generating each possible combination for attribute dimensions (increasing sequences), and makes the selections consistent to these dimensions. Treating equivalent sets up to permutation as one single case, we get the number of different types (cases). The output of the program is a grouped presentation in terms of the spreadsheet representation.

8.2.2 The Ternary Case

The total number of closed sets given three fixed attributes is 45. If permutation of attributes does not matter, we get the number of different types of ternary relationships which is 14. Table 8.2 shows the spreadsheet representation of the sets with their generating systems and attribute dimensions indicated.[1]. These sets were also presented in [17] but a different numbering system is used here, based on the ordering by attribute dimensions.

Note the dimension class ($[A] = 1,\ [B] = 2,\ [C] = 2$) is missing since it is a contradictory case. It is easily verified that no valid set of functional dependencies correspond to this combination of attribute dimensions. If $\mathcal{F}$ were such a set, $[B] = 2$ and $[C] = 2$ would imply $AC \to B,\ AB \to C \in \mathcal{F}$ and no one-dimensional functional dependency would determine B or C. Similarly, $[A] = 1$ would imply a one-dimensional constraint

[1]The triangular graphical presentation of the sets can be found in Section 9.1

#	BC → A	AC → B	AB → C	B → A	A → B	C → A	A → C	C → B	B → C	Generating system of functional dependencies	Dimension of attributes $[A]$	$[B]$	$[C]$
0	0	0	0	0	0	0	0	0	0	$\emptyset$	∞	∞	∞
1	1	0	0	1	0	0	0	0	0	$\{B \to A\}$	1	∞	∞
2	1	0	0	1	0	1	0	0	0	$\{B \to A, C \to A\}$	1	∞	∞
3	1	1	0	0	0	1	0	1	0	$\{C \to B, C \to A\}$	1	1	∞
4	1	1	0	1	0	1	0	1	0	$\{C \to B, B \to A\}$	1	1	∞
5	1	1	0	1	1	0	0	0	0	$\{A \to B, B \to A\}$	1	1	∞
6	1	1	0	1	1	1	0	1	0	$\{C \to B, A \to B, B \to A\}$	1	1	∞
7	1	1	1	1	1	0	1	0	1	$\{A \to C, A \to B, B \to A\}$	1	1	1
8	1	1	1	1	1	1	1	1	1	$\{A \to B, B \to C, C \to A\}$	1	1	1
9	1	1	1	0	0	1	0	1	0	$\{AB \to C, C \to B, C \to A\}$	1	1	2
10	1	1	0	1	0	0	0	0	0	$\{AC \to B, B \to A\}$	1	2	∞
11	1	0	0	0	0	0	0	0	0	$\{BC \to A\}$	2	∞	∞
12	1	1	0	0	0	0	0	0	0	$\{BC \to A, AC \to B\}$	2	2	∞
13	1	1	1	0	0	0	0	0	0	$\{BC \to A, AC \to B, AB \to C\}$	2	2	2

Table 8.2: The sets of functional dependencies for the ternary case, grouped by dimensions of attributes

determining A is in $\mathcal{F}$. Assume $B \to A \in \mathcal{F}$. Since $\{B \to A,\ AB \to C\} \models B \to C$, $B \to C \in \mathcal{F}$ ($\mathcal{F}$ is a closed set). This is a contradiction since $B \to C$ is a one-dimensional constraint determining C and so $[C] = 1$ would hold. Similar contradiction with B arises by assuming $C \to A \in \mathcal{F}$.

8.2.3 Quaternary and Quinary Cases

Similarly to the ternary case, quaternary and quinary cases can be generated and counted.

For the quaternary case, the total number of sets is 2 271, treating equivalent sets as one case we get 165 cases. These are listed in the spreadsheet form in Appendix B.1.

As the number of attributes is raised to five, the total number of sets increases to more than a million (exactly 1 373 701) while the different types remain relatively low, 14 480. A summarization of the number of sets by attribute dimension groups is presented in B.2. Table 8.3 shows the cases of class ($[A] = 1, [B] = 1, [C] = 1, [D] = 2, [E] = 3$) as an example. Grouped bits of the binary (tabular) representation represent the presence or absence of dependencies in the following order (from left to right):

$(BCDE \to A,\ ACDE \to B,\ ABDE \to C,\ ABCE \to D,\ ABCD \to E)$,
$(BCD \to A,\ ACD \to B,\ ABD \to C,\ ABC \to D)$,
$(BCE \to A,\ ACE \to B,\ ABE \to C,\ ABC \to E)$,
$(BDE \to A,\ ADE \to B,\ ABE \to D,\ ABD \to E)$,
$(CDE \to A,\ ADE \to C,\ ACE \to D,\ ACD \to E)$,
$(CDE \to B,\ BDE \to C,\ BCE \to D,\ BCD \to E)$,
$(BC \to A,\ AC \to B,\ AB \to C)$, $(BD \to A,\ AD \to B,\ AB \to D)$,
$(BE \to A,\ AE \to B,\ AB \to E)$, $(CD \to A,\ AD \to C,\ AC \to D)$,
$(CE \to A,\ AE \to C,\ AC \to E)$, $(DE \to A,\ AE \to D,\ AD \to E)$,
$(CD \to B,\ BD \to C,\ BC \to D)$, $(CE \to B,\ BE \to C,\ BC \to E)$,

$(DE \to B,\ BE \to D,\ BD \to E),\ (DE \to C,\ CE \to D,\ CD \to E),$
$(B \to A,\ A \to B),\ (C \to A,\ A \to C),\ (D \to A,\ A \to D),\ (E \to A,\ A \to E),$
$(C \to B,\ B \to C),\ (D \to B,\ B \to D),\ (E \to B,\ B \to E),$
$(D \to C,\ C \to D),\ (E \to C,\ C \to E),\ (E \to D,\ D \to E).$

#	Binary representation
...	
$[A]=1,[B]=1,[C]=1,[D]=2,[E]=3$	
1068	11111 1000 0110 1110 1110 1101 000 100 010 100 010 110 000 110 100 100 00 00 10 00 00 00 10 00 10 00
1069	11111 1001 0111 1110 1110 1101 000 100 010 100 010 110 000 110 100 100 00 00 10 00 00 00 10 00 10 00
1070	11111 1001 0111 1110 1110 1101 000 100 010 101 010 110 000 110 100 100 00 00 10 00 00 00 10 00 10 00
1071	11111 1100 1110 1110 0111 0111 110 110 110 000 010 010 000 010 010 100 11 00 00 00 00 00 00 00 10 00
1072	11111 1001 0111 1110 1110 1101 000 101 010 101 010 110 000 110 100 100 00 00 10 00 00 00 10 00 10 00
1073	11111 1100 1110 1110 1111 1111 110 110 110 000 010 110 000 010 110 100 11 00 00 00 00 00 00 00 10 00
...	

Table 8.3: An example class of the quinary case with 6 sets of functional dependencies

8.2.4 Summary of the Number of Closed Sets

Denote by $\mathcal{SD}_n$ the set of closed sets of functional dependencies for a relation schema with n attributes (with constant attributes disallowed). Furthermore, let τ the equivalence relation on these sets classifying them into different cases (for two equivalent sets there exists a permutation of attributes transforming one set to another). The number of different classes ($\mathcal{SD}_n/\tau$) exactly correspond to the number of relationship types if the attributes do not have a fixed role. If we allow attributes to be stated as constants, it yields a larger set that is exactly the set of Moore families [46] for n, denoted by $\mathcal{SD}_n^0$ and its equivalence classes $\mathcal{SD}_n^0/\tau$. For each $n \in \mathbb{N}^+$, $\left|\mathcal{SD}_{n+1}^0/\tau\right| = \left|\mathcal{SD}_{n+1}/\tau\right| + \left|\mathcal{SD}_n^0/\tau\right|$ easily follows, as well as $\left|\mathcal{SD}_n^0\right| = \sum_{i=0}^{n}\binom{n}{i}\left|\mathcal{SD}_i\right|$ where $\left|\mathcal{SD}_0\right| = 1$.

With these notations, Table 8.4 shows the number of different cases for known arities and demonstrates the combinatorial of the search space. The first five rows were computed by the PROLOG program (see Insert) and the third column was also obtained by [48]. The number of Moore families for six elements was presented in [46] and the first column can be calculated from that by the summarization formula above. The number of different equivalence classes for the sixth row is still unknown, as well as the numbers for higher arities[2].

8.3 Spreadsheet Reasoning

To use the spreadsheet for reasoning, the notation must be extended to allow the distinction for the state of constraints as follows.

Let 1 and 0 indicate the functional dependencies and excluded functional constraints of the initial set, respectively. All others are marked with '.' initially, indicating that their state is not known. As we get an implied positive constraint (functional dependency) during the deduction process, the corresponding '.'

[2]Estimations exist, see [14, 30]

Table 8.4: Number of closed sets of functional dependencies for n attributes. The column printed in italics contains the number of cases with no constant attributes of arity n (equivalent with the basic relationship types)

n	$\lvert\mathcal{SD}_n\rvert$	$\lvert\mathcal{SD}_n/\tau\rvert$	$\lvert\mathcal{SD}^0_n\rvert$	$\lvert\mathcal{SD}^0_n/\tau\rvert$
1	1	*1*	2	2
2	4	*3*	7	5
3	45	*14*	61	19
4	2 271	*165*	2 480	184
5	1 373 701	*14 480*	1 385 552	14 664
6	75 965 474 236	*?*	75 973 751 474	?

BC → A	AC → B	AB → C	B → A	A → B	C → A	A → C	C → B	B → C	Implication impact of detected functional constraints
.	⊢ 1	.	.	1	.	.	.	.	(S) $A \rightarrow B \vdash AC \rightarrow B$
.	.	1	.	1	.	⊢ 1	.	.	(T) $AB \rightarrow C,\ A \rightarrow B \vdash A \rightarrow C$
.	0	.	.	⊢ 0	.	.	⊢ 0	.	(P) $AC \not\rightarrow B \vdash A \not\rightarrow B,\ C \not\rightarrow B$
.	.	⊢ 0	.	1	.	0	.	.	(Q) $A \rightarrow B,\ A \not\rightarrow C \vdash AB \not\rightarrow C$
.	.	1	.	⊢ 0	.	0	.	.	(R) $AB \rightarrow C,\ A \not\rightarrow C \vdash A \not\rightarrow B$

Table 8.5: Example of the spreadsheet derivation of functional constraints. Rules of the PQRST implication system in the spreadsheet form

is replaced with '⊢1' if the state of the implied constraint was previously unknown. Similarly, an implied negated constraint is indicated by '⊢0'.

The spreadsheet may be used for deriving contradictions as well. Contradictions occur whenever new constraints are introduced and the implication system allows to derive the opposite. Contradiction can be indicated by the symbol $\lightning$.

The STRPQ algorithm presented in Section 7.3.2 provides a possible way for derivation of the full knowledge a partial set holds in terms of the spreadsheet representation. As an example, Table 8.5 shows how rules of the PQRST implication system (see Section 7.3) can be represented in the spreadsheet form for the ternary case. Further examples can be found in [31, 34].

Chapter 9

Graphical Reasoning on Sets of Functional Constraints

9.1 Triangular Representation

A set of functional constraints can be represented by a diagram. Functional dependencies and excluded constraints are indicated as nodes of geometrical figures. Graphical reasoning is based on the graphical interpretation of the PQRST ruleset presented in Section 7.3.

9.1.1 The Triangular Representation for the Ternary Case

The triangular representation for three attributes in shown on Figure 9.1. The diagram consists of a triangle and three separate edges. Nodes of the triangle are placeholders of two-dimensional constraints. The right-hand side of the constraint determines the location in the triangle. Similarly, nodes of the separately drawn edges correspond to one-dimensional constraints. This notation matches the concept of constraint dimension because the triangle is geometrically a two-dimensional shape and the edges are one-dimensional.

Validity of a constraint is denoted by a circle. The initial set of constraints (basic constraints) are denoted by filled circles. All other nodes are empty, indicating their state is unknown yet. During constraint set development, implied constraints are denoted by empty circles. After a closure is reached, the remaining constraints can be specified as basic (either positive or negative) and derivation can be continued.

Depicting a closed set of functional dependencies need the indication of positive constraints only. Negated constraints may be needed during constraint acquisition. If both positive and negative constraints are considered, the following notations are used in the figures:

Basic functional dependencies are denoted by filled circles.

Implied functional dependencies are denoted by empty circles.

Negated basic functional dependencies are denoted by crossed filled circles.

Implied negated functional dependencies are denoted by crossed empty circles.

Figure 9.2 shows some examples of the triangular representation. All different ternary cases of closed sets (relationship types) are shown on Figure 9.3.

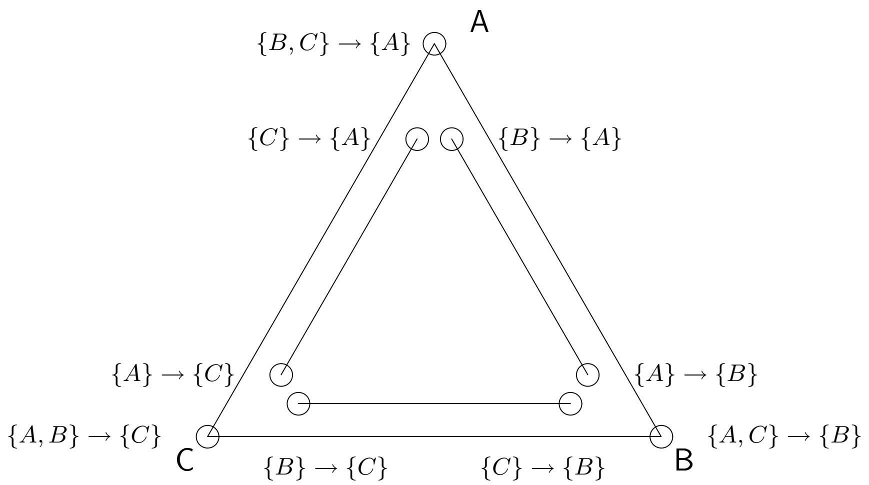


Figure 9.1: Triangular representation of sets of functional dependencies for the ternary case

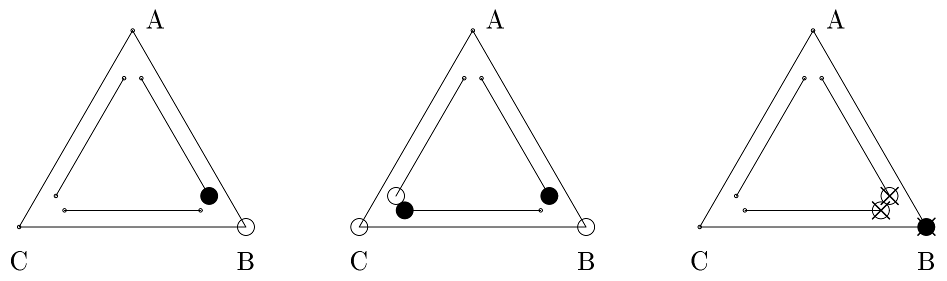


Figure 9.2: Examples of the triangular representation. From left to right: 1. The functional dependency $\{A\} \to \{B\}$ and the implied functional dependency $\{A,C\} \to \{B\}$. 2. The functional dependencies $\{A\} \to \{B\}$, $\{B\} \to \{C\}$ and their implied functional dependencies. 3. The negated functional dependency $\{A,C\} \not\to \{B\}$ and the implied negated functional dependencies $\{A\} \not\to \{B\}$ and $\{C\} \not\to \{B\}$

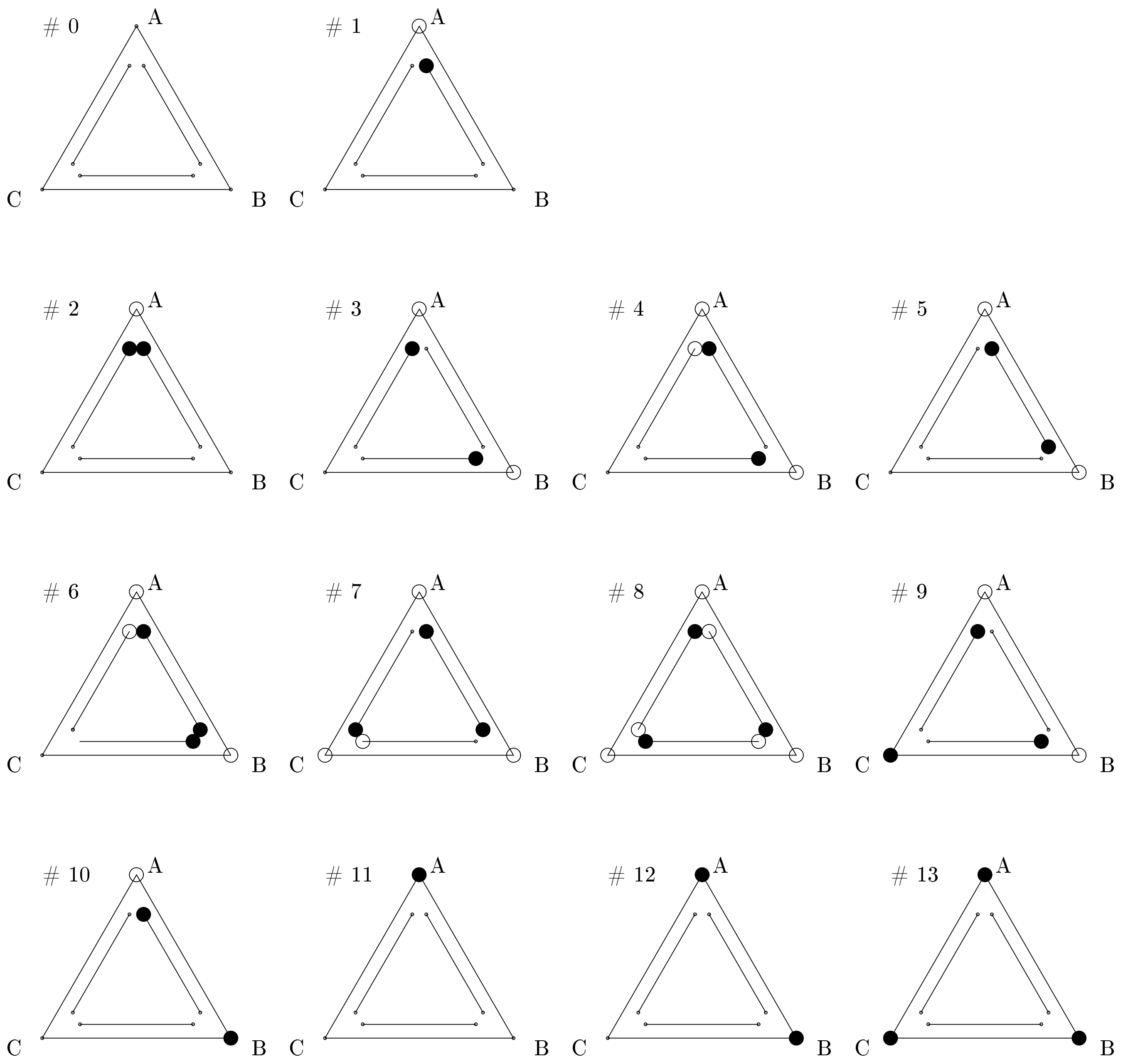


Figure 9.3: All sets of functional dependencies in ternary relationship types

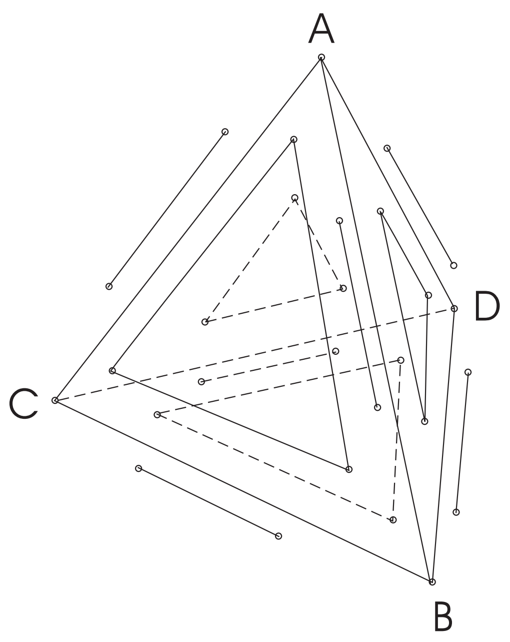


Figure 9.4: A tetrahedron as the 3D graphical representation for four attributes (stripped lines indicate invisible edges from the front)

9.1.2 The Quaternary Case

As mentioned above, the triangular representation can be generalized to higher number of attributes. Generalization can be performed in two directions: representation in a higher-dimensional space (3D in the case of 4 attributes) or constructing a planar (2D) representation.

Consider the triangular representation for the ternary case. It consists of two nested binary cases (represented by edges) It is obvious that the quaternary case contains four nested ternary cases (triangles) with their one-dimensional parts shared (6 edges). Additionally, three-dimensional constraints can be represented as vertices of a three-dimensional shape which is actually a tetrahedron. This way we get a representation in 3D space, where each node is a placeholder of a functional dependency or excluded constraint (see Figure 9.4). For better visibility, separate edges are drawn outside the tetrahedron where possible.

Substituting the 3-dimensional tetrahedron with a square and rearranging the other elements yields a planar, quadratic representation presented on Figure 9.5.

Section 8.2.3 presented the quaternary constraint sets of the attribute dimension class ($[A] = [B] = 1$, $[C] = [D] = 2$). It is presented in the graphical form on Figure 9.6 as an example for both graphical representations.

9.1.3 The Quinary Case

The higher-dimensional representation for 5 attributes exists in the 4D space with 5 nested quaternary cases, 10 ternary and 10 binary cases. Each of the edges (corresponding to a binary case) has 3 neighboring ternary cases (which the binary case is nested in) and 3 neighboring quaternary cases, while each of the triangles (corresponding to a ternary case) has 2 neighboring quaternary cases. The frame of the four-dimensional

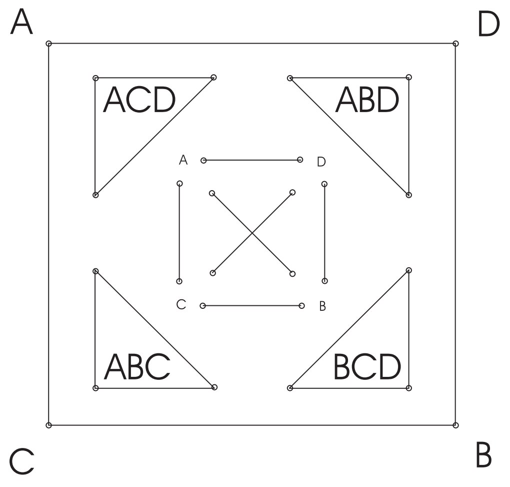


Figure 9.5: The quadratic representation

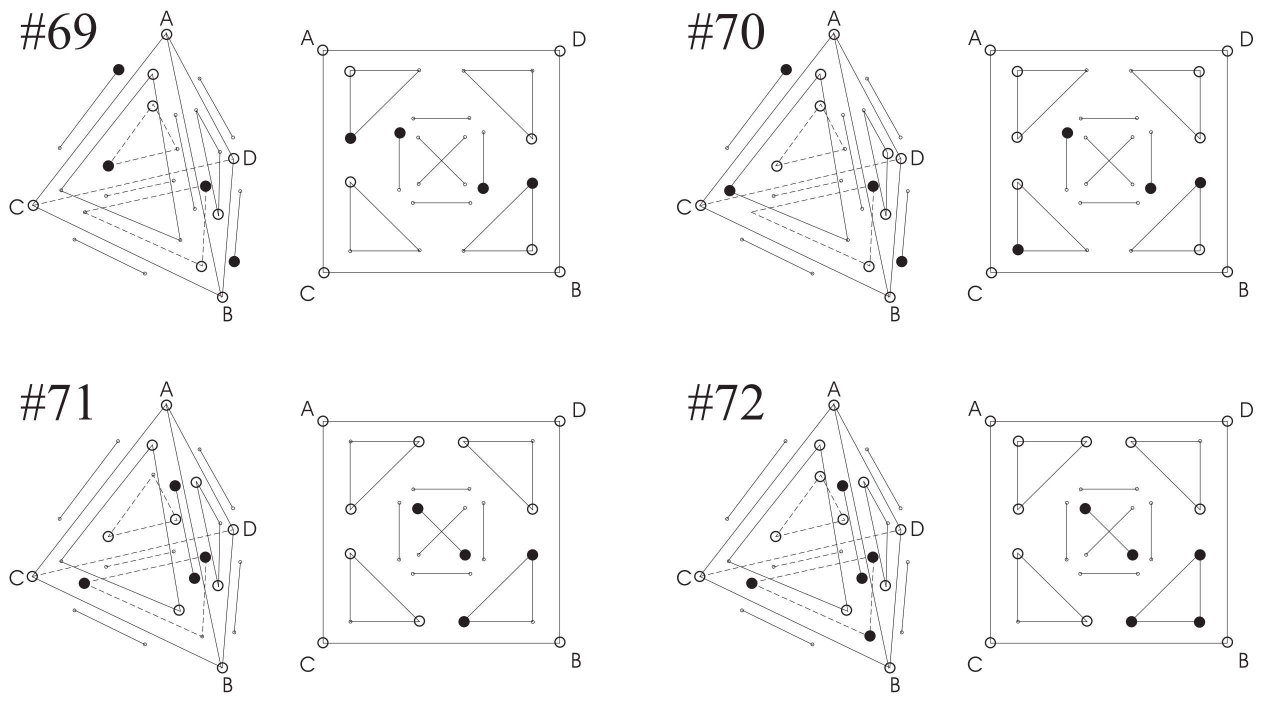


Figure 9.6: The tetrahedral and quadratic graphical representations of sets with $[A] = [B] = 1$, $[C] = [D] = 2$ (numbers of sets refer to the spreadsheet representation presented in Section 8.2.3)

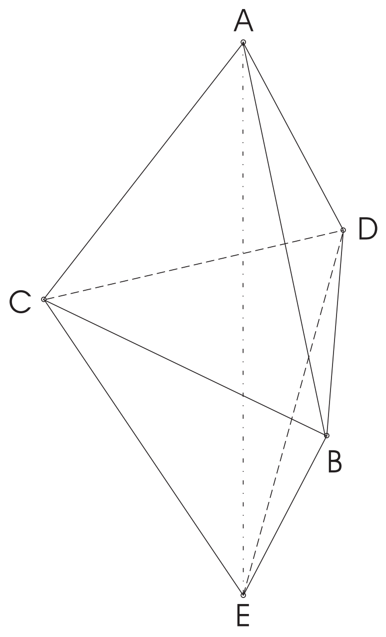


Figure 9.7: A three-dimensional projection of the frame of the four-dimensional object for the representation of FD sets over 5 attributes. The five tetrahedra corresponding to nested quaternary cases can easily be discovered, sharing their surface triangles (they represent the ten nested ternary cases) and edges (ten nested binary cases).

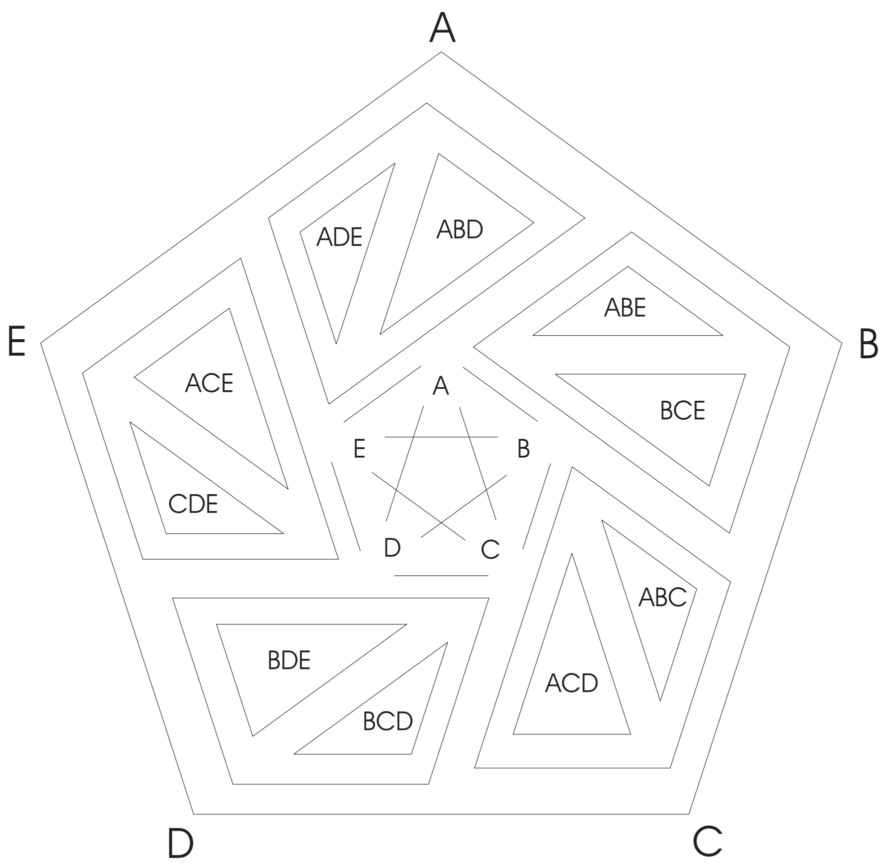


Figure 9.8: The pentagonal representation for sets of functional dependencies over 5 attributes

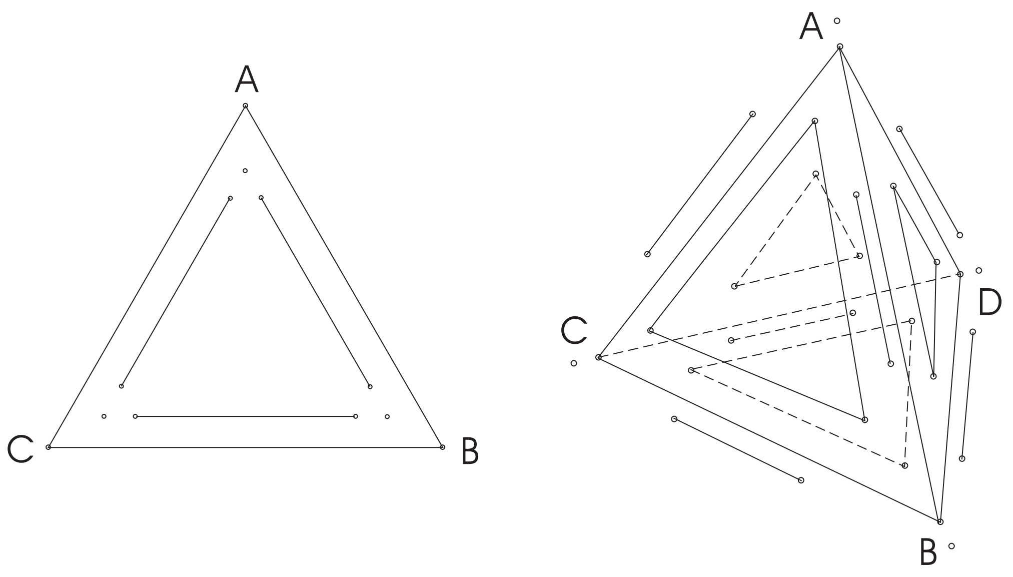


Figure 9.9: Extending the triangular and tetrahedral representations with placeholders of zero-dimensional constraints as separate vertex nodes

object is shown on Figure 9.7 as projected to three dimensions. To get the full representation, 5 tetrahedra, 10 triangles and 10 edges formed by the nodes of attributes should be added separately to the figure.

A two-dimensional version can be constructed the same way as the quadratic representation for the quaternary case and is shown on Figure 9.8. Each node of the graph is a placeholder of a functional constraint (dependency or negated constraint) placeholder. Each trapezoid corresponds to a tetrahedron and the bounding pentagon corresponds to the whole body in the 4-dimensional representation. The number of constraint placeholders is 75. One can easily discover which 3 edges belong to a specific triangle or which 4 triangles belong to a specific trapezoid (or corresponding tetrahedron) by looking at the parallel lines and directions of attributes.

9.2 Possible Generalizations

9.2.1 Incorporating constant attributes

For a complete representation of sets of functional constraints, it may be necessary to incorporate dependencies of the form $\emptyset \rightarrow A$ into the system where A is an attribute, although they are usually not needed for data modeling issues. Extending the graphical representation is straightforward, making the geometrical analogy complete by incorporating the vertices themselves as separately drawn zero-dimensional components denoting zero-dimensional constraints (see Figure 9.9 for the extended triangular and tetrahedral representations).

9.2.2 Increasing the number of attributes

For a number of attributes n, a generalized triangular representation can be constructed in the $n-1$ dimensional space.Although these higher-dimension representations may be used as data structures for storing

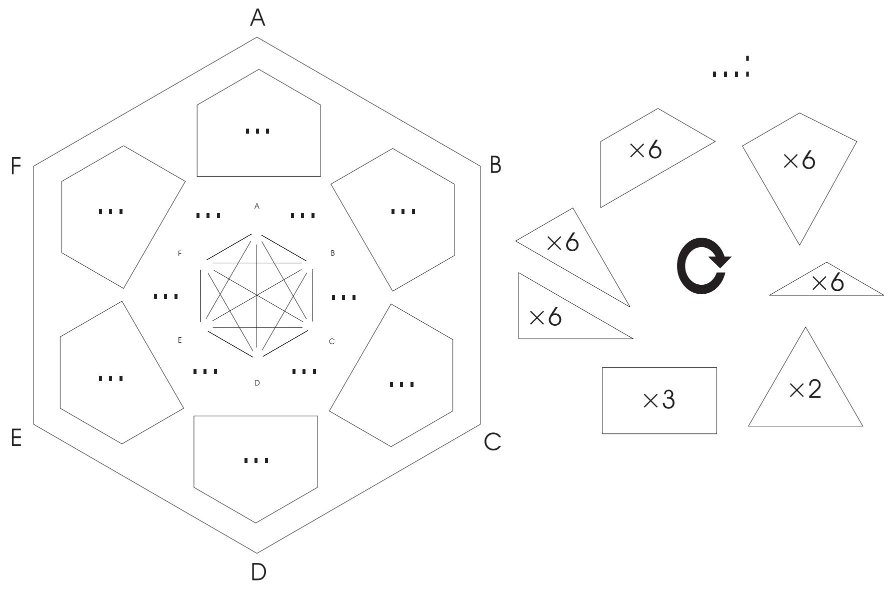


Figure 9.10: Towards the hexagonal representation as a possible generalization for 6 attributes. The frame on the left-hand side is to be extended by rotations of components on the right-hand side

sets in memory, their graphical presentation becomes rather complex for human perception as the number of attributes is raised.

For instance, the generalized triangular representation for 6 attributes exists in 5D space. A possible way of constructing a non-redundant 2D representation is shown on Figure 9.10. A case of six attributes has 6 nested quinary cases (4D-objects or pentagons), 15 nested quaternary cases (tetrahedra or quadrangles), 20 ternary and 15 binary cases (triangles and edges, respectively). The total number of constraint placeholders (nodes) is 186.

If we replace the notion of attributes with entities, this also illustrates the contrast between a complete description of a relationship with higher arity and the usual notation of an entity-relationship graph. Certainly, most of the relationship types (sets of functional dependencies) occur rarely in practice since most relationships with higher arity can be decomposed. However, it is important to note that normalization produces a good result only if the specification of constraints is complete, so such a representation may make sense before normalization. In practice, this usually depends on the separation of concepts (entities, relationships) but a premature separation may cause the loss of some 'hidden' dependencies.

9.3 Graphical Rules and Reasoning

Recall the implication systems ST and PQRST with their rules (P), (Q), (R), (S) and (T) from section 7.3. These rules can be interpreted in terms of the graphical representation as well. A deduction step using one of them deals with a node of a higher-dimension object (e.g. triangle as a two-dimensional object with one of its three vertices) and one or two of its borders (with one dimension lower, e.g. edges of the same triangle as one-dimensional objects).

Graphical versions of rules are shown on Figure 9.11 for the triangular representation (case $Y = \{C\}$).

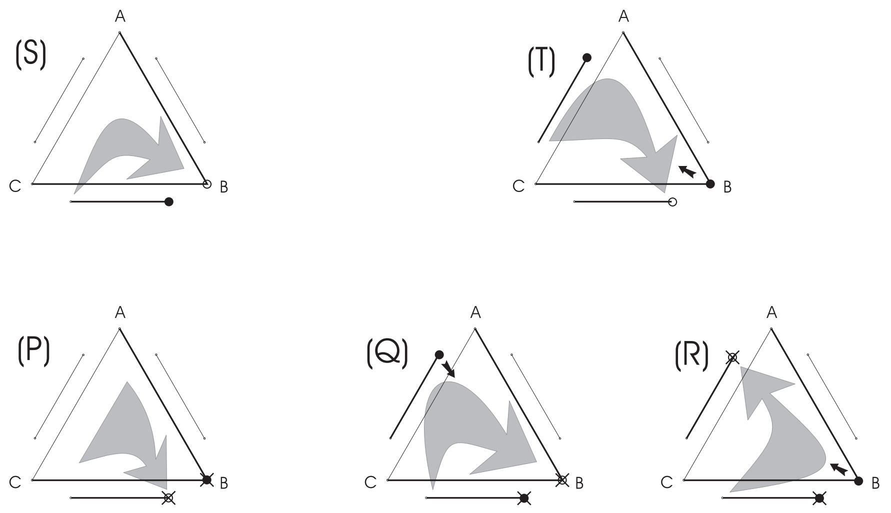


Figure 9.11: Graphical versions of rules (P), (Q), (R), (S) and (T) in terms of the triangular representation. The small black arrows indicate support (necessary context) while the large grey arrows show the implication effects

The large grey arrows indicate the implication effect of each rule. Rule (S) is a simple extension rule and rule (T) can be called as "rotation rule" or "reduction rule". We may call the left-hand side of a functional dependency the *determinant* of it and the right-hand side the *determinate*. Rule (S) can be used to extend the determinant of a dependency resulting another dependency, while rule (T) is used for *rotation*, that is, to replace the determinate of a functional dependency by the support of another functional dependency with one dimension higher (the small black arrow at B indicates support of $AC \to B$). Another possible way to interpret rule (T) is for reduction of the determinant of a higher-dimensional dependency by omitting an attribute if a dependency holds among the attributes of the determinant.

For excluded functional constraints, rule (Q) acts as the extension rule (needs support of a positive constraint, i.e. functional dependency) and (R) as the rotation rule (needs a positive support too). These two rules can also be viewed as negations of rule (T). Rule (P) is the reduction rule for excluded functional constraints, with the opposite effect of rule (Q) (but without the need of support). Rule (Q) is also viewed as the negation of rule (S).

Recall Figure 9.2 from Section 9.1 for three examples of the ternary case. The triangle on the left-hand side shows an example of the application of graphical (triangular) rule (S). On the right-hand side, rule (Q) is used twice while in the middle one rule (S) is used twice followed by (T). The middle one also demonstrates that no explicit rule for transitivity is needed.

These graphical rules can be generalized to higher dimensional representations, where the number of attributes is more than 3. Figure 9.12 illustrates the rules (S) and (T) in terms of the tetrahedral representation. Each of the rules has two versions, with $|Y| = 1$ and $|Y| = 2$. Rules for excluded constraints can be generalized the same way. This generalization can be carried on to larger schemata with raising the possible sizes of left-hand sides of constraints and so, extending figure 9.12 with additional columns of higher-dimensional operations.

Figure 9.13 shows the patterns of rules (S) and (T) for the quadratic representation. The reason why two or three patterns arise for a single case is the broken symmetry of the quadratic representation compared to

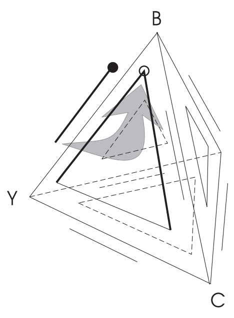
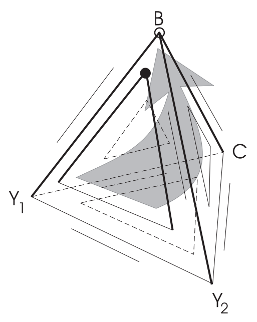
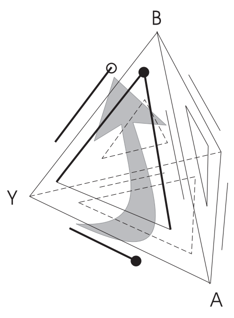
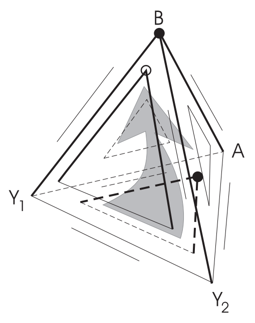
	$\|Y\|=1$	$\|Y\|=2$
(S)		
(T)		

Figure 9.12: Patterns of graphical rules (S) and (T) for different sizes of Y, presented in the tetrahedral representation

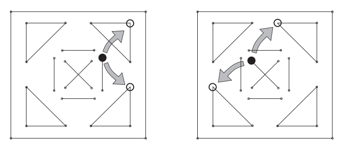
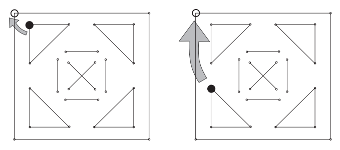
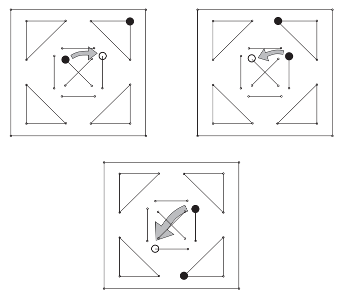
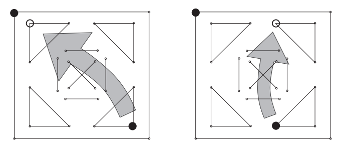
	|Y|=1	|Y|=2
(S)		
(T)		

Figure 9.13: Patterns of graphical rules (S) and (T) for the quadratic representation

the tetrahedron (diagonals and sides of the square are conceptually equivalent edges).

The ST algorithm presented in Section 7.3.2 can be used with the triangular, tetrahedral, quadratic or other representations. Figure 9.12 (Section 9.3) can also be viewed as a table of elementary operations used by this algorithm in the quaternary case. As an example, Figure 9.14 shows how the complete set presented as quaternary case #64 in Appendix B.1 can be derived from a generating system using the ST algorithm. The STRPQ algorithm can be used with negated constraints.

Using rule (U) as many times as possible may be an alternative algorithm for positive constraints since the system U is sound and complete as well (see Section 7.4.2). This way we have more types of possible graphical 'operations' (10 patterns) and selecting the preconditions is usually more complex but we may derive dependencies in less number of steps.

The structure of the generalized triangular representations (triangular, tetrahedral, etc.) may also be used for designing a data structure representing sets of functional constraints for the algorithms.

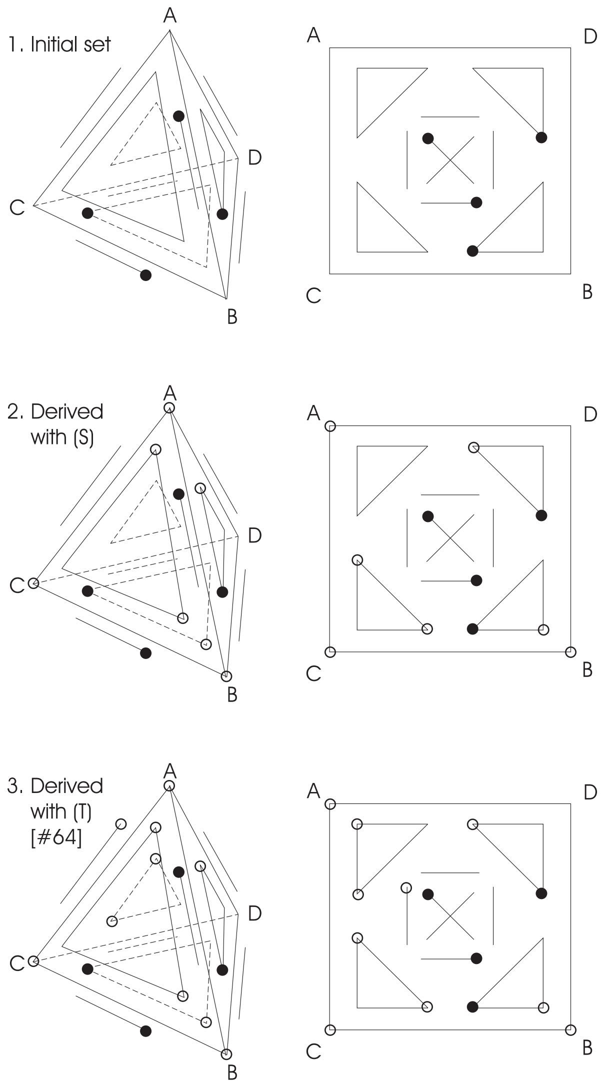


Figure 9.14: The ST algorithm used for deriving dependencies of case #64 (see Appendix B.1) starting with its generating system

Chapter 10

Further Methods on Sets of Functional Constraints

This chapter extends the previous chapters on representation and reasoning for functional constraints. Methods presented in this section go beyond the task of deriving all implied constraints, further tasks of managing constraint sets are considered such as insertion or conversion.

10.1 Managing Constraint Sets

10.1.1 Inserting or Deleting a Functional Constraint into a Closed Set

We can now derive from a set of given constraints all constraints that are implied and that are contradicted. We, thus, obtain a number of constraints whose validity is still open. Using the approach of [4] we can generate sample data and provide them to the designer with the question whether these data support a certain functional dependency or not. At this point, a new constraint is inserted and only the new implications needed to be generated, involving a contradiction check. Sets of functional dependencies can be definitively developed this way.

Based on the PQRST method (Section 7.3.2), the following recursive algorithm adds a (positive) functional dependency to a closed set:[1]

```
$Add(\mathcal{F}, A_1 A_2 \dots A_k \to B)$ :

    if $A_1 \dots A_k \not\to B$ then $\lightning$, stop
    else if $A_1 \dots A_k \to B$ then return
    else declare $A_1 \dots A_k \to B$ as implied and
      1. (S): $\forall C : Add(\mathcal{F}, A_1 \dots A_k C \to B)$
      2. (T): $\forall C$ : if $A_1 \dots A_k B \to C$ then $Add(\mathcal{F}, A_1 \dots A_k \to C)$
      3. (T support): $\forall i$ : if $A_1 \dots A_{i-1} A_{i+1} \dots A_k \to A_i$ then
         $Add(\mathcal{F}, A_1 \dots A_{i-1} A_{i+1} A_k \to B)$
```

[1] Universe of quantification $\forall C$ contains all the attributes except $A_1 \dots A_k$ and B. Universe of quantification $\forall i$ is $[1 \dots k]$. Quantifications can be implemented by *for* loops.

```
        4. (R support): $\forall i$ : if $A_1 \dots A_{i-1}A_{i+1} \dots A_k \not\twoheadrightarrow B$ then
           $Add(\mathcal{F}, A_1 \dots A_{i-1}A_{i+1}A_k \not\twoheadrightarrow A_i)$
        5. (Q support): $\forall C$ : if $A_1 \dots A_k \not\twoheadrightarrow C$ then
           $Add(\mathcal{F}, A_1 \dots A_kB \not\twoheadrightarrow C)$
      end if
  end.
```

To insert a negated constraint, the algorithm looks like this:

```
  $Add(\mathcal{F}, A_1A_2 \dots A_k \not\twoheadrightarrow B)$ :
      if $A_1 \dots A_k \to B$ then ↯, stop
      else if $A_1 \dots A_k \not\twoheadrightarrow B$ then return
      else declare $A_1 \dots A_k \not\twoheadrightarrow B$ as implied and
        1. (R): $\forall C$ : if $A_1 \dots A_kC \to B$ then $Add(\mathcal{F}, A_1 \dots A_k \not\twoheadrightarrow C)$
        2. (Q): $\forall C$ : if $A_1 \dots A_k \to C$ then $Add(\mathcal{F}, A_1 \dots A_kC \not\twoheadrightarrow B)$
        3. (P): $\forall i$ : $Add(\mathcal{F}, A_1 \dots A_{i-1}A_{i+1}A_k \not\twoheadrightarrow B)$
      end if
  end.
```

The algorithms work by searching all possible consequences and support cases of each constraint recursively based on the PQRST implication system. Their efficiency is proportional to the number of newly implied constraints multiplied by the number of attributes of the schema. It is due to the simplicity of the rules: the number of possible instantiations given one of their prerequisite constraints (i.e. when Y and B fixed) is bounded by the number of attributes – one free parameter remains which is an attribute (e.g. C for rule (S) and A for rule (T)). This is not true for the Armstrong implication system. Since there are constant number of rule application types after adding a single constraint (5 for a positive and 3 for a negative constraint), complexity of the whole recursive algorithm is bounded by a constant (five or three) times the number of attributes multiplied by the total number of added constraints.

Deleting a constraint could be performed by a same type of recursion. However, an implied constraint cannot be removed so the first step is to check whether it is implied by other constraints. This should be done for all implications as well. Since mutual implications may exist, deletion can only be carried over by a two-phase method: first flag all the implied constraints in the spreadsheet that can be derived using the one to be deleted and then check which of them can be derived using other, remaining constraints. This may cause some flags to be removed (some constraints not to be deleted). After this procedure is completed, flagged constraints can be removed from the constraint set and the set remains closed.

10.1.2 Closing an Attribute Set

If the set is not closed and one wants to know the closure of an attribute set X, three methods arise using the graphical (or spreadsheet) representation, each having strong connection with the well-known closure algorithm. The first step is always dropping dependencies whose right-hand side is in X. Afterwards, attributes are added to X one-by-one. The methods may be combined.

Closing by paths needs the graph to be treated as a hypergraph with dependencies as hyper-edges.[2] To get the closure of an attribute set, one seeks for all attributes that can be reached by a hyper-path starting from X.

Closing by extension means we initially generate dependencies in the form $X \to A$ from each $Y \to A, Y \subset X$ by rule (S) and collect them into a set $\mathcal{F}_X$. Then X is extended by A and $X \to A$ is dropped. In the next step, dependencies of $\mathcal{F}_X$ are extended with A by (S) to match the new X and other dependencies are added from the initial set whose left sides are equal to the new X. This is repeated until $\mathcal{F}_X$ is empty.

Closing by translation is based on the method discussed in [26] adapted to the graphical representation. As an initial step, all basic dependencies are translated by formally removing each attribute of X from the left-hand sides. Some dependencies become zero-dimensional, they can be represented on the graph as extra nodes at the attributes. This way we get a reduced schema and construct the graphical (or spreadsheet) representation of it. It is especially useful if the schema has many attributes but the initial set X is relatively large too. We represent only the difference of them. The constant attributes of the reduced schema – denoted by Y – are in the closure. The next step is a further reduction of the schema by removing attributes of Y and drawing a new graph. The new dependencies are obtained the same way as before, formally removing elements of Y from the dependencies (it means we simply generate the reductions of them). This process is repeated until no further attributes exist (X is a key) or we get an empty Y (X is not a key).

10.1.3 Validity of a Single Functional Dependency

For a non-closed set of dependencies, there are basically two options to look for the validity of a specific dependency – just like in any reasoning system.

A straightforward way is to close the attribute set on the left-hand side of the desired dependency using one of the methods discussed above and see whether the attribute on the right-hand side is in the closure.

As an alternative, a target-oriented method can be achieved by declaring the desired constraint as the goal and generate an *and/or structure* of sub-goals the goal can be derived from – e.g. reductions of the functional dependency are alternative sub-goals since if one of them holds, the goal is achieved by the extension rule (S). This method is less efficient, especially if the desired constraint turns out not to be true. However, reversing this gives a possibility of *abductive reasoning*: knowing the goal is true as a basic dependency, the designer can seek for possible causes and decide whether or not to reconsider this basic constraint as implied from some newly found and declared basic dependencies ('explanations') and fine-tune the set of constraints by them.

10.2 Conversion Between Representations

The semilattice approach, presented in Section 2.3.4 an an alternative means of representation of closed functional dependency sets has a great significance in the partition model for describing the refinement structure of partitions (structural graph, see Chapter 5). This section considers conversion between the semilattice representation and the simplified traditional formalism, which the triangular and spreadsheet representations are based on.

[2]A hyper-edge can have more start points (corresponding to the left side of a dependency) but only one endpoint (the right side).

10.2.1 Obtaining the Closed Attribute Sets from a Closed Set of Functional Dependencies

Whether or not a specific attribute set is closed can be seen from the spreadsheet representation of a closed constraint set in the dimension group corresponding to the size of the attribute set. To construct the semilattice graph of all the closed sets needs systematically collecting the closed sets. It is performed on the basis of the following rules:

1. The set of all attributes is always closed.
2. If a set X is closed and no dependency $Y \to A$ holds where $Y \subset X$ and $A \in X \setminus Y$ then Y is closed.
3. If two sets X and Y are closed then $X \cap Y$ is closed.

Start with the full set with size n (first rule). Then look at each attribute sets with size $n-1$ whether the dependency holds whose left-hand side is that set (simply displayed as the highest dimension group in the spreadsheet). If the dependency does not hold, the set is added to the closed attribute sets (second rule). Then we add the intersections of the obtained sets as closed sets (third rule). This process is repeated with the possible sets in decreasing order by their sizes. The third rule ensures once a closed set is found, determining whether a subset of it is closed needs only to check existence of dependencies completely inside (and not pointing outside) of the set. The whole process can be done also by hand for small relation schemata using the spreadsheet representation.

10.2.2 Transforming the Semilattice Graph into a Closed Set of Functional Constraints

Given the semilattice of the closed attribute sets, negated functional constraints can be obtained: if a set X is closed, then $X \not\to A$ holds for each $A \notin X$. These are declared as initial constraints in the dimension group of the spreadsheet corresponding to the size of X. All other negated constraints can be derived afterwards by the negated reduction rule (P). All the remaining nodes correspond to positive functional dependencies.

An example of conversion can be found in [34].

10.3 The Impact of Sets of Functional Dependencies

Assume a closed set of functional dependencies is given as a description of a relationship in the conceptual model. The designer needs to know whether the schema needs decomposition, how to decompose it and how to transform it to a relational database schema with all the necessary constraints (see also Section 2.3.5). Discussion of these questions for different possible sets of functional dependencies may give interesting and useful results.

If the schema is in a suitable normal form then decomposition is usually not necessary. Two types of normal forms are relevant if functional dependencies are considered [42]. The following definitions are given in terms of the simplified syntax.

A schema is in *BCNF (Boyce–Codd normal form)* if the left side of each positive constraint contains a subset that is key (i.e. the set of dependencies can be generated by key dependencies). This normal form provides a nonredundant schema but it is not always possible to reach without the loss of some dependencies.

A generalization of BCNF is the *3NF (Third normal form).* A schema is in 3NF if for each positive constraint the left side contains a key or the right side is contained by a key. It is well-known that for each possible functional dependency set there exists a dependency-preserving 3NF decomposition.

Case #	Camps # [17]	Key(s)	BCNF	3NF	Possible BCNF decomposition
#0	#0	ABC	x	x	(no)
#1	#7	BC	-	-	AB, BC
#2	#11	BC	-	-	AB, AC, BC
#3	#6	C	x	x	(AC, BC)
#4	#9	C	-	-	AB, BC
#5	#8	AC/BC	-	x	AB, BC or AB, AC
#6	#10	C	-	-	AB, BC or AB, AC
#7	#5	A/B	x	x	(AB, BC, AC)
#8	#4	$A/B/C$	x	x	(several)
#9	#13	AB/C	x	x	(no)
#10	#12	AC/BC	-	x	no!
#11	#3	BC	x	x	(no)
#12	#2	AB/BC	x	x	(no)
#13	#1	$AB/BC/AC$	x	x	(no)

Table 10.1: Normalization and the sets of functional dependencies for the ternary case

The process of normalization can be automated by well-known algorithms. However, there are cases with several possibilities when the designer can take the best choice, taking into account other issues in the application (e.g. the semantics of attributes). In some cases several BCNF decompositions are possible, while some others are already in BCNF but can be decomposed further. Or another case to mention is: when a 3NF decomposition is reached, is it worth further decomposing into BCNF (if possible)? And is there any chance to get a a non-redundant decomposition without loss of constraints if there is no equivalent BCNF decomposition?

[17] shows for each ternary relationship type (closed set of functional dependencies) how it can be transformed into relational database schema. It reveals the need of more complex constraints to get an equivalent schema with the original relationship type (e.g. inclusion dependencies on natural joins), even if it is decomposed into a normal form.

Table 10.1 gives a brief analysis on the possible ternary cases. The first column refers to the numbering used in Table 8.2 (Section 8.2.2) and in Figure 9.3 (Section 9.1). The second column corresponds to the numbering used by [17]. The third column lists keys, the fourth and fifth contains an 'x' if the schema is in BCNF or 3NF, respectively. The last column contains possible BCNF decompositions of the schema. If the schema is already in BCNF then it is put in brackets because it is less relevant.

Note that case # 10 is the only ternary one that has no dependency-preserving BCNF decomposition. By the way, it is already in 3NF.

The cases can be classified according to their decomposability properties:

No decomposition can be applied: In terms of the triangular representation, the connectivity in the graphs of the functional dependencies does not allow a decomposition. Either the components are not associated to each other or are heavily inter-twined with each other. These schemata must be in 3NF but not necessarily in BCNF:

- Cases # 0, # 9, # 11, # 12 and # 13 are in BCNF. A further decomposition is not even necessary.

- There does not exist a dependency-preserving decomposition into a BCNF or a non-redundant decomposition in a 3NF in case # 10 (we would repeat the group A, B again in the other component). The designer has to decide whether to leave the schema in redundant 3NF or to decompose it to BCNF with the loss of some dependencies (and reformulate the lost dependencies as special constraints in the database schema). However, there may be other solutions for splitting the schema: [62] considered the `city` (A), `ZIP` (B) and `street` (C) example and proposed to split the ZIP component into two, e.g. ZIP-digits-for-city (B_1) and Additional-ZIP-digits (B_2). In this case we obtain the system $\{B_1 \to A,\ A \to B_1,\ AC \to B_2\}$ and a dependency-preserving BCNF decomposition becomes possible as either (B_1B_2C, AB_1) or (AB_2C, AB_1).

Decomposition is possible but may not be preferable: Decomposition can be applied and a split into two relationship types is possible. However, the schema is already in normal form. The designer can decide whether a further decomposition is reasonable or not.

- Cases # 3, # 7, and # 8 are in BCNF, therefore a further decomposition is usually not preferable.
- Case # 5 is in 3NF but not in BCNF. It is up to the designer whether a further decomposition is useful, depending on the application.

Decomposition is preferable: The schema is not in normal form and can be split into two components. In such cases a decomposition is preferable but may not be unique. There may be cases when 3NF decomposition is possible but no dependency-preserving BCNF can be reached (although such patterns do not exist for the ternary case).

- The decomposition is unique in cases # 1, # 2, and # 4 and yields a BCNF schema.
- Case # 6 allows two different BCNF decompositions. Which one is the most appropriate, depends on the application.

This survey shows that the impact of functional dependencies is far not trivial, even with the consideration of well-studied normal forms. There are 'interesting patterns' that should be observed and treated with care in real applications. Ternary cases #5, #6 and #10 are examples of such patterns. Their most suitable decomposition for the application needs the interaction of the designer. Moreover, [16] has shown that the common belief is wrong that says each relationship type described by a set of functional dependencies can be transformed into an equivalent relational schema using key dependencies and (possibly non-key based) inclusion dependencies. Some patterns (#2, #10 for ternary) need more sophisticated constraints.

Graphical or spreadsheet reasoning can be used to reach a complete specification of a relationship type. We may use these patterns for reasoning on decompositions and transformation into relation schemata.

10.4 Future Issues

Implementation of the discussed methods can give a software tool support for designing relationships by sets of functional constraints, helping effective database design.

Future work may include the construction of a spreadsheet or graphical method for other types of constraints as well, like cardinality, multivalued or inclusion dependencies. They have similar characteristics as functional dependencies (but some types are not finitely axiomatizable). Multivalued dependencies denote concept independence and can be represented by their dependency bases. There are some interesting, competing (or conflicting) cases [74].

It is still open for six attributes how many different cases of functional dependency sets exist (the number of cases up to permutation of attributes). For more than 6 attributes, only estimations exist [14, 30]. In fact, a deeper analysis of the known cases (quaternary, quinary) is also promising in order to have more sophisticated reasoning facilities on the conceptual level on types of relationships. Surveying different sets of functional constraints for higher arities may reveal more interesting patterns. A first step towards that is made by the simplified axiomatization and by generating and counting the possible closed sets.

Part IV

Summary

Chapter 11

Summary and Conclusion

11.1 The General Partition Data Model

This thesis proposed the foundations of an application-independent framework for modeling multiple partitions over possibly different base sets, on one hand. The starting point is the reformulation of the conjunctive part of the logical-deductive partition model of Spyratos [71, 70]. The model is extended with the concept of aggregation (cumulative) attributes. With this special semantics, the well-known generic relational data model can represent finite set systems, manage the set names and handle summarizable aggregation data, based on the principle of disjointness. A tuple in a relation denotes an elementary set intersection.

Naive usage of relational algebra can be misleading and the operations do not always give a valid result with partition semantics. Base set relationships must be taken into account. A sound partition algebra is presented together with base set constraints, which determine the validity of operations. Base set constraints restrict the applicable partition semantics. Since the semantics are not part of the - strictly taken - partition database, they are considered as metadata. Some of them have a syntactical aspect, which is a necessary and sufficient condition for the existence of proper semantics.

Refinement structure of partitions can be declared by functional dependencies. Closed sets of attributes correspond to the different possible partitions. A non-closed set represents the same elementary set intersections as its closure. The semilattice of closed sets yields a graph representation of the partition structure (structural graph). Paths of the graph correspond to the possbile aggregation levels. It corresponds to the traditional lattice of cuboids but more complex refinement structures can be described. The structural graph notation can be extended for the case of partially applicable attributes (sub-partitioning) and incomparable partitions. There is a semi-automatic method for creating a partition relational database schema given a structural graph. User navigation operations can be formulated as base set and attribute selection in terms of the structural graph. It is translated into partition algebra via expressions attached to the nodes of the structural graph.

The general partition data model has formally four levels: the structural, the logical administrative, the instance and the interpretation level. These levels are more or less similar to the schema and instance level of the traditional relational approach. The structural level gives a seamless navigation over the conceptual partition structure, based on partition attributes and base sets. This allows any suitable decomposition, including redundant relations (e.g. materialized views) can be used without affecting the higher level operations. The logical level contains the relational schema and the attached algebra expressions to the structural graph. Instance level stands for the actual partition relational data and resolving of set names is the matter of interpretation level – which is supposed to be external information by default. Several basic cases are given for demonstrating the power of the model and for motivating the operations of a basic partition language. Currently, specification of the operations sufficient for the described base cases is given. Some parts of the model and the language can be defined as 'plug-ins', i.e. those parts can be changed to meet the application needs.

Although the framework is in its first phase yet, it is capable to model a simple multidimensional schema as a special case and furthermore, complex dependencies inside a dimension, heterogenity or missing data. It partially solves the first and third problem discussed in [49]. A partition relation is especially useful to describe sparse multiway partitions (e.g. a sparse data cube or a multiple partitioning of geographic space). If the information available is not complete, the model allows to merge the available data into a unified structure and one may determine exactly what can be computed and what not.

The complete foundations must be worked out, based on real-world examples. Important issues are the effects of the levels to each other, the impacts of local dependencies in detail, data manipulation, specification of plug-ins and aggregation generalization. Extension and refinement possibilities may also be investigated to meet criteria introduced in [65] and [51]. Detailed research on base set constraint implication and the effect on constraints of the algebra operations is still a future issue. More sophisticated data modeling tools will provide a convenient schema design, taking into account what data is available and what information should the database provide.

The general partition database model is a promising platform for laying the formal foundations of data warehouses, including aggregation semantics, with proper extension of the model.

11.2 Representation and Reasoning on Functional Constraints

The refinement structure of partitions can be desribed by a closed set of functional dependencies. This motivated a more detailed investigation of constraint sets, on the other hand. The results can be used outside the partition model too, in relational database theory and data modeling.

Methods are proposed for reasoning on sets of functional constraints using simple and sophisticated graphical and spreadsheet representation, which help a database designer to achieve complete and unambiguous specification of schemata, especially relationships of higher arity.

The traditional conceptual modeling languages, such as E/R have restricted capabilities for specification of higher arity relationships and do not support logical implication. Unambiguity of specification is a prerequisite of schema refinement and normalization. At the same time, the traditional formalism of functional dependencies have inherent redundancies (e.g. trivial constraints). All this leads to a inconvenience - and sometimes ignorance - of precise constraint acquisition.

The main focus is on considering sets of constraints as a whole instead of constraints one-by-one as in the traditional functional dependency notation. Since constraint acquisition is an iterative process of acquiring the validity or invalidity of constraints one-by-one, negated (excluded) dependencies are used to explicitly allow specification of invalidity of functional dependencies. Inherent redundancy of the traditional syntax is eliminated by not considering trivial and nonsingleton constraints. It requires a different axiomatization that is a simple and powerful implication system PQRST convenient for graphical and spreadsheet representations is taken as a basis for reasoning. Soundness and completeness are shown. Moreover, there exists a specific order of rule application to derive all implied dependencies: can be described using the regular expression $(S)^*; (T)^*; (R)^*; ((P)\|(Q))^*$. This can be fine-tuned by the dimension of constraints (the size of their left-hand sides). Other tasks have also considered in terms of this reasoning framework, including conversion between this type of representation and the semilattice of closed attribute sets (the structural graph notation of partitions). All these point towards the future implementation of these methods in a software tool.

Based on the ST axiomatization, the different possible sets are generated for up to 5 attributes [31]. We know the number of possible sets for 6 fixed attributes from [46]. It is still unknown for higher arities. Another possible future direction is surveying the known cases more deeply. Seeking such representation methods for other types of constraints (e.g. multivalued, cardinaliy constraints) seems a wide field of future development.

Since database design approaches rely on completeness and soundness of constraint sets, all these may help database designers to obtain better database design results.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases.* Addison-Wesley, Reading, MA, 1995.

[2] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases.* Addison-Wesley Publishing Company, 1995.

[3] Rakesh Agrawal, A. Gupta, and Sunita Sarawagi. Modeling multidimensional databases. In Alex Gray and Per-Åke Larson, editors, *Proc. 13th Int. Conf. Data Engineering, ICDE*, pages 232–243. IEEE Computer Society, 7–11 1997.

[4] M. Albrecht, E. Buchholz, A. Düsterhöft, and B. Thalheim. An informal and efficient approach for obtaining semantic constraints using sample data and natural language processing. In *Proc. Semantics in Databases, LNCS 1358*, pages 1–28. Springer, Berlin, 1998.

[5] W. W. Armstrong. Dependency structures of data base relationships. In J. L. Rosenfeld, editor, *Information Processing 74, Proceedings of IFIP Congress 74*, pages 580–583, Stockholm, Aug. 5-10,1974, 1974. North-Holland, Amsterdam.

[6] P. Atzeni and R. Torlone. A metamodel approach for the management of multiple models and the translation of schemes. *Information Systems*, 18(6):349–362, 1993.

[7] G. Birkhoff. *Lattice Theory.* Amer. Math. Soc., Providence, R. I., 1940. First edition, Colloquium Publications.

[8] J. Biskup. Boyce-codd normal forma and object normal forms. *Information Processing Letters*, 32(1):29–33, 1989.

[9] J. Biskup. *Foundations of information systems.* Vieweg, Wiesbaden, 1995. In German.

[10] J. Biskup, J. Demetrovics, L. O. Libkin, and M. Muchnik. On relational database schemes having a unique minimal key. *J. of Information Processing*, 27:217–225, 1991.

[11] J. Biskup and T. Polle. Decomposition of database classes under path functional dependencies and onto contraints. In *Proc. FoIKS'2000*, LNCS 1762, pages 31–49. Springer, 2000, 2000.

[12] P. De Bra and J. Paredaens. Horizontal decompositions and their impact on query solving. *SIGMOD Rec.*, 13(1):46–50, 1982.

[13] Paul De Bra. Functional dependency implications, inducing horizontal decompositions. In Joachim Biskup, János Demetrovics, Jan Paredaens, and Bernhard Thalheim, editors, *MFDBS*, volume 305 of *Lecture Notes in Computer Science*, pages 80–98. Springer, 1987.

[14] G. Burosch, J. Demetrovics, G. O. H. Katona, D. J. Kleitman, and A. A. Sapozhenko. On the number of databases and closure operations. *TCS*, 78(2):377–381, 1991.

[15] Luca Cabibbo and Riccardo Torlone. Integrating heterogeneous multidimensional databases. In James Frew, editor, *SSDBM*, pages 205–214, 2005.

[16] R. Camps. From ternary relationship to relational tables: A case against common beliefs. *ACM SIGMOD Record, 31(2)*, pages 46–49, 2002.

[17] R. Camps. Transforming n-ary relationships to database schemas: An old and forgotten problem. Technical Report LSI-5-02R, Universitat Politècnica de Catalunya, 2002.

[18] N. Caspard and B. Monjardet. The lattices of closure systems, closure operators, and implicational systems on a finite set: a survey. *Discrete Applied Mathematics*, 127:241–269, 2003.

[19] Paul Chan and Arie Shoshani. Subject: A directory driven system for large statistical databases. In Harry K. T. Wong, editor, *SSDBM*, pages 61–62. Lawrence Berkeley Laboratory, 1981.

[20] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[21] Chen & Associates, Baton Rouge, LA. *ER-designer reference manual*, 1986–1989.

[22] P. P. Chen. The entity-relationship model: Toward a unified view of data. *ACM TODS*, 1(1):9–36, 1976.

[23] P. P. Chen, editor. *Proc. 1st Int. ER Conf., ER'79: Entity-Relationship Approach to Systems Analysis and Design*, Los Angeles, USA, 1979, 1980. North-Holland, Amsterdam.

[24] E. F. Codd. A relational model for large shared data banks. *CACM*, 13(6):197–204, 1970.

[25] S. Cosmadakis, P. Kanellakis, and N. Spyratos. Partition semantics of relations. *Journal of Computer and System Sciences*, 33(2), 1986.

[26] J. Demetrovics and N. X. Huy. Translations of relation schemes and representations of closed sets. *PU.M.A.Ser. A*, 1(3-4):299–315, 1990.

[27] J. Demetrovics and G. O. H. Katona. Combinatorial problems of database models. In *Colloquia Mathematica Societatis Janos Bolyai 42, Algebra, Combinatorics and Logic in Computer Science*, pages 331–352, Györ, Hungary, 1983.

[28] J. Demetrovics, L. Libkin, and I. B. Muchnik. Functional dependencies and the semilattice of closed classes. In *Proceedings of MFDBS 89, Lecture Notes in Computer Science 364*, pages 136–147. Springer Verlag, Berlin, Germany, 1989.

[29] J. Demetrovics, L. Libkin, and I. B. Muchnik. Functional dependencies in relational databases: A lattice point of view. *Discrete Applied Mathematics*, 40(2):155–185, 1992.

[30] J. Demetrovics, L. O. Libkin, and I. B. Muchnik. Functional dependencies and the semilattice of closed classes. In *Proc. MFDBS'89, LNCS 364*, pages 136–147, 1989.

[31] J. Demetrovics, A. Molnar, and B. Thalheim. Graphical and spreadsheet reasoning for sets of functional dependencies. Technical Report 0404, Kiel University, Computer Science Institute, http://www.informatik.uni-kiel.de/reports/2004/0404.html, 2004.

[32] J. Demetrovics, A. Molnar, and B. Thalheim. Graphical reasoning for sets of functional dependencies. In *Proceedings of ER 2004, Lecture Notes in Computer Science 3288*, pages 166–179, Shanghai, China, 2004. Springer Verlag.

[33] J. Demetrovics, A. Molnar, and B. Thalheim. Graphical reasoning for sets of functional dependencies. In *Proceedings of ER 2004, Lecture Notes in Computer Science 3288*, pages 166–179. Springer Verlag, 2004.

[34] J. Demetrovics, A. Molnar, and B. Thalheim. Relationship design using spreadsheet reasoning for sets of functional dependencies. In *Proceedings of ADBIS 2006, Lecture Notes in Computer Science 4152*, pages 108–123, Thessaloniki, Greece, 2006. Springer Verlag.

[35] J. Demetrovics, A. Molnár, and B. Thalheim. Relációs adatbázisok funkcionális függõségeinek grafikus axiomatizációja. *Alkalmazott matematikai lapok*, 2007, Budapest (accepted for publication).

[36] J. Demetrovics, A. Molnár, and B. Thalheim. Graphs representing sets of functional dependencies. *Annales Univ. Sci. Budapest, Sectio Computatorica*, 28, 2008, (accepted for publication).

[37] J. Demetrovics and V. D. Thi. Some results on functional dependencies. *Acta Cybernetica*, 8(3):273–278, 1988.

[38] Jan Van den Bussche. The semijoin algebra. In Jürgen Dix and Stephen J. Hegner, editors, *FoIKS*, volume 3861 of *Lecture Notes in Computer Science*, page 1. Springer, 2006.

[39] M. Erwig and M. Schneider. Partition and conquer. In *Spatial Information Theory*, pages 389–407, 1997.

[40] M. Erwig and M. Schneider. The honeycomb model of spatio-temporal partitions. In *Spatio-Temporal Database Management*, pages 39–59, 1999.

[41] J. Lisboa F., V. F. Sodré, J. Daltio, M. F. Rodrigues Jr., and V. Vilela. A case tool for geographic database design supporting analysis patterns. In *Proceedings of of Conceptual Modeling for Advanced Application Domains. 1st Int. Workshop on Conceptual Modeling for GIS (CoMoGIS – ER2004), Lecture Notes in Computer Science 3289*, pages 43–54, Shanghai, China, 2004. Springer Verlag.

[42] H. Garcia-Molina, J. Ullman, and J. Widom. *Database systems: The complete book.* Prentice Hall, Upper Saddle River, 2002.

[43] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alexander S. Szalay, David DeWitt, and Gerd Heber. Scientific data management in the coming decade, 2005.

[44] Jim Gray and Alexander S. Szalay. Where the rubber meets the sky: Bridging the gap between databases and science, 2004.

[45] M. Gyssens and L. V. S. Lakshmanan. A foundation for multi-dimensional databases. In *The VLDB Journal*, pages 106–115, 1997.

[46] N. Habib and L. Nourine. The number of moore families on n=6. *Discrete Mathematics*, 294(3):291–296, 2005.

[47] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Academic Press, Morgan Kaufmann Publishers, San Diego, USA, 2001.

[48] A. Higuchi. Note: Lattices of closure operators. *Discrete Mathematics*, 179:267–272, 1998.

[49] W. Hmmer, W. Lehner, A. Bauer, and L. Schlesinger. A decathlon in multidimensional modeling: Open issues and some solutions. In *DaWaK 2000: Proceedings of the 4th International Conference on Data Warehousing and Knowledge Discovery*, pages 275–285, London, UK, 2002. Springer-Verlag.

[50] John Horner, Il-Yeol Song, and Peter P. Chen. An analysis of additivity in olap systems. In *DOLAP '04: Proceedings of the 7th ACM international workshop on Data warehousing and OLAP*, pages 83–91, New York, NY, USA, 2004. ACM Press.

[51] C. S. Jensen, A. Kligys, T. B. Pedersen, and I. Timko. Multidimensional data modeling for location-based services. *The VLDB Journal*, 13(1):1–21, 2004.

[52] D. Laurent and N. Spyratos. Partition semantics of relations. In *Proceedings of PODS'85 (ACM-SIGACT-SIGMOD Symposium on Principles of Database Systems)*, Oregon, USA, 1985.

[53] D. Laurent and N. Spyratos. Introducing negative information in relational databases. In *Proceedings of MFCS'88 (Mathematical Foundations of Computer Science), Lecture Notes in Computer Science 324*, Karlsbad, Chechoslovakia, 1988.

[54] D. Laurent and N. Spyratos. Partition semantics for incomplete information in relational databases. In *Proceedings of ACM-SIGMOD Conference*, Chicago, USA, 1988.

[55] D. Laurent and N. Spyratos. A partition approach to updating universal scheme interfaces. *IEEE Transactions on Knowledge and Data Engineering*, 6(2), 1994.

[56] C. Lecluse and N. Spyratos. Incorporating functional dependencies in deductive query answering. In *Proceedings of ICDE'87 (IEEE International Conference on Data Engineering)*, Los Angeles, USA, 1987.

[57] Wolfgang Lehner, Jens Albrecht, and Hartmut Wedekind. Normal forms for multidimensional databases. In *Proceedings of 10th SSDBM*, pages 63–72, Capri, Italy, 1998.

[58] H-J. Lenz and A. Shoshani. Summarizability in OLAP and statistical data bases. In *Statistical and Scientific Database Management*, pages 132–143, 1997.

[59] H-J. Lenz and B. Thalheim. Olap databases and aggregation functions. In *Proceedings of 13th SSDBM*, pages 91–100, Fairfax, Virginia, 2001.

[60] H-J. Lenz and B. Thalheim. Warning: Cube may mislead. In *Proceedings of CSIT'06*, volume 2, pages 7–16, Amman, Jordan, 2006.

[61] L. Libkin. Direct product decompositions of lattices, closures and relation schemes. *Discrete Mathematics*, 112:119–138, 1993.

[62] J. A. Makowsky and E. V. Ravve. Dependency preserving refinements and the fundamental problem of database design. *DKE*, 24(3):277–312, 1998. Special Issue: ER'96 (ed. B. Thalheim).

[63] T. Martyn. Reconsidering multi-dimensional schemas. *SIGMOD Rec.*, 33(1):83–88, 2004.

[64] C.W. Morris. Foundations of the theory of signs. In *International Encyclopedia of Unified Science*. University of Chicago Press, 1955.

[65] T. B .Pedersen and C. S. Jensen. Multidimensional data modeling for complex data. Technical report TR-37, TimeCenter, 1998.

[66] P. Revesz, editor. *Introduction to Constraint Databases*. Springer Verlag, New York, USA, 2002.

[67] P. Rigaux and M. Scholl. Multi-scale partitions: Application to spatial and statistical databases. In *Symposium on Large Spatial Databases*, pages 170–183, 1995.

[68] P. Rigaux, M. Scholl, and A. Voisard. *Spatial databases with application to GIS*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[69] H. Sato. Handling summary information in a database: Derivability. In Y. Edmund Lien, editor, *SIGMOD Conference*, pages 98–107. ACM Press, 1981.

[70] N. Spyratos. The partition model: A functional approach. Technical Report 430, INRIA, Centre de Rocquencourt, 1985.

[71] N. Spyratos. The partition model: A deductive data base model. *ACM Transactions on Database Systems*, 12(1):1–37, 1987.

[72] B. Thalheim. Open problems in relational database theory. *Bull. EATCS*, 32:336 – 337, 1987.

[73] B. Thalheim. *Entity-relationship modeling – Foundations of database technology*. Springer, Berlin, 2000. See also http://www.informatik.tu-cottbus.de/~thalheim/HERM.htm.

[74] B. Thalheim. Conceptual treatment of multivalued dependencies. In *Proceedings of ER 2003, Lecture Notes in Computer Science 2813*, pages 363–375. Springer Verlag, 2003.

[75] V. D. Thi. Minimal keys and antikeys. *Acta Cybernetica*, 7:361–371, 1986.

[76] Panos Vassiliadis. Modeling multidimensional databases, cubes and cube operations. In *Statistical and Scientific Database Management*, pages 53–62, 1998.

[77] C.-C. Yang. *Relational Databases*. Prentice-Hall, Englewood Cliffs, 1986.

Part V

Appendices

Appendix A

Basic Partition Language Operations

This Appendix lists proposed operations of the basic partition language for the general partition model presented in Chapter 6. Refer to Section 6.1 for levels of the framework and Section 6.2.1 for types used in signatures of operations. The syntax is not fixed: there is a plug-in facility in the model for set symbol language as well as set, partition and aggregation definition expression formulae. The currently established syntax is based on the plug-in specifications in the referred sections.

This illustrative collection of operations are motivated by the sample cases in Section 6.3. It can be treated as a type specification for partition databases: if these operations are available then the sample cases can be modeled. Semantics of operations are given informally, together with hints for future implementation.

At the current stage it is preferred to attach cumulative attributes to all partition attributes of a base set and to attach all functional dependencies to the primary base set of their attributes. The reason is the impact of local dependencies and partial validity of aggregation is not fully studied yet.

More operations will be necessary for a complete language (e.g. construction and selection operators for each type used in the language, iteration operator for set type construction, etc), which shall be designed by taking into account more sophisticated real-life examples and reasonable extensions to the model (see Section 6.5).

A.1 Partition Database Construction Operations

Construction operations can be used to define or extend the partition database schema and to populate it with data. Data manipulation framework must be developed in the future and its operations will be needed for completeness.

The actual partition database is an implicit input-output parameter of the operations in this section.

A.1.1 Structural Level Declarations

The following operations work on the structural level and are used for partition structure declaration.

InitSchema($SetSymb\, u$**)**

Initializes the partition database schema. The superset of all sets (*universal base set*, root set) will be denoted by u. The second step of construction must be the partitioning of u.

DeclareAlias($\mathit{SetSymb}\,a$, $\mathit{SetPath}\,h$**)**

This is a proposal of a syntactic sugar additionally to the database schema. Set name a is created as a global alias for set h (namespaces may also be used). Symbol a can be used as a shorthand of set path h.

DeclarePartition($\mathit{SetPath}\,h$, $\mathit{PartStruct}\,\mathcal{G}$**)**

Declares a partition structure for base set h by the structural graph $\mathcal{G}$. Structural graph construction operations can be found in Section A.3. $\mathcal{G}$ contains partition attributes and their refinement structure together with cumulative attributes. This defines the valid partitions. The known partitions will be a subset of this, determined by logical level operations.

Set h must be a valid, existing set. If it already has a partition structure then this declaration must extend it without contradiction. If an existing cumulative attribute is given then its cumulative operation cannot be changed. Previous functional dependencies must not be invalidated, but new dependencies may be added to the existing set (in terms of structural graph, no new nodes containing only existing attributes can be introduced but existing nodes can be dropped).

The declaration will be checked against the actual data (known partitions). If the data violates the new structure then this operation fails.

A.1.2 Partition Construction

These operations work on the structural as well as the logical level. They declare new partitions and construct them from existing data in the database.

There are basically three ways for constructing a partition from existing data:

- *Partition derivation* takes attributes of the same base set and constructs a new partition based on them. Some disjoint sets can be merged but atomic set intersections cannot be split (it would need external knowledge and cannot be performed by derivation).
- *Partition inheritance.* Any subset that is selected from a partition automatically inherits the attributes of its parent (superset) with their derivations, constraints, etc. No extra operations are needed.
- *Attribute construction (unification).* Disjoint subsets of a base set is taken and their own partitions are put together to form a partition of the base set.

A special combination of these methods is the construction of *characteristic partition* of a subset w.r.t. a superset. This partition contains two subsets, one corresponds to the given subset and the other one to its complement regarding the superset.

DerivePartition($\mathit{PartSetPath}\,h[A]$, $\mathit{Partition}[\alpha]$, $\mathit{AttrDef}(A)\,\delta$**)**

The new partition attribute A with base set h is defined based on partition $h[\alpha]$. Constraint $\alpha \to A$ and the attribute derivation formula will be added to the structure.

Partition $h[\alpha]$ must be known. New nodes of the structural graph on the logical level corresponding to partitions known in combination with $h[\alpha]$ will be automatically labeled with algebra expressions generated from the attribute definition δ, using the expression attached to the node of $h[\alpha]$.

Case-switch conditions and assignments in the attribute definition must refer to $h[\alpha]$ only. If attribute A does not exists (i.e. it is not valid), the operation calls DeclarePartition. If it already exists, the the derivation is declared and verified as a consistency check.

Section 6.3.5 presents an example.

ConstructPartition($\mathit{PartSetPath}\, h[B]$, $\mathit{PartSymb}\, A$, $\mathit{AttrDef}(B)\, \delta$**)**

The new partition $h[B]$ is constructed from partitions of disjoint subsets specified by partition $h[A]$. Case-switch conditions of definition δ must be atomic equalities of attribute A with constants (additionally, the special condition *else*). The assignment(s) must refer to constant values or attributes of h or the appropriate subset[1] only.

Partition $h[A]$ must be known. New nodes containing B in the structural graph are checked whether they are known and algebra expressions are assigned similarly to DerivePartition. An example is presented in Section 6.3.8.

ConstructAndUnifyPartitions($\mathit{PartSetPath}\, h[A]$, $\mathit{PartSymb}\, B$, $\mathit{AttrDef}(B)\, \delta$**)**

Similar to ConstructPartition but δ cannot contain constant values (except in the *else* branch) or concatenation. Partition attributes of the disjoint subsets used for construction are unified and renamed as B. The result is like B being declared on h and inherited by the subsets.

General Set Operations

Special cases of ConstructPartition. New partition $h[A]$ is created with two subsets a and $\bar{a}$. Set a represents the constructed set out of k and l (union, intersection or difference of them) and $\bar{a}$ its complement.

Sets k and l must be subsets of h so that their intersection and differences can be constructed: they must be components of partitions of h whose combination is known.

ConstructUnionPartition($\mathit{PartSetPath}\, h[A]$, $\mathit{SetSymb}\, a$, $\mathit{SetSymb}\, \bar{a}$, $\mathit{SetPath}\, k$, $\mathit{SetPath}\, l$**)** :
$h[A].a = k \cup l$

ConstructIntersectPartition($\mathit{PartSetPath}\, h[A]$, $\mathit{SetSymb}\, a$, $\mathit{SetSymb}\, \bar{a}$, $\mathit{SetPath}\, k$, $\mathit{SetPath}\, l$**)** :
$h[A].a = k \cap l$

ConstructDiffPartition($\mathit{PartSetPath}\, h[A]$, $\mathit{SetSymb}\, a$, $\mathit{SetSymb}\, \bar{a}$, $\mathit{SetPath}\, k$, $\mathit{SetPath}\, l$**)** :
$h[A].a = k \setminus l$

ConstructSimmDiffPartition($\mathit{PartSetPath}\, h[A]$, $\mathit{SetSymb}\, a$, $\mathit{SetSymb}\, \bar{a}$, $\mathit{SetPath}\, k$, $\mathit{SetPath}\, l$**)** :
$h[A].a = k \Delta l$

An example can be found in Section 6.3.9 on how ConstructUnionPartition can be performed using ConstructPartition. Others follow the similar way.

ConstructCharacteristicPartition($\mathit{PartSetPath}\, h[A]$, $\mathit{SetSymb}\, k$, $\mathit{SetSymb}\, \bar{k}$, $\mathit{SetPath}\, l$**)**

A characteristic partition of set l is created in h as $h[A]$: value k of A denotes $h \cap l$ and value $\bar{k}$ denotes $h \setminus l$.

If l is a 'direct subset' of h, i.e. it can be expressed using the known partitions of h then the characteristic partition is easily constructed using ConstructPartition. If it can be reached via a subset path from h then ConstructPartition is called recursively. Known partitions in combination with A are determined by ConstructPartition. If l and h are disjoint, A will have the constant value $\bar{k}$. If their relationship is not known, the operation fails.

[1]For a condition $A = a$ attributes of $h[A].a$ can be used.

DropPartition($PartSetPath\, h[\alpha]$**)**

Deletes the given partition (with its data if known). The operation fails if the partition is referenced by a construction or derivation or automatically inherited.

DereferencePartition($PartSetPath\, h[\alpha]$**)**

If the partition is referenced by constructed or derived attributes then this operation removes these references (the derived or constructed attributes become unknown and can be externalized).

RenamePartition($PartSetPath\, h[\alpha], Partition\, [\beta]$**)**

Attributes of α are renamed as β based on positions with all of their references. If the attributes are inherited the operation fails. This operation may also be useful for moving names across namespaces.

A.1.3 Partition Data Assignment

Logical level data assignments follow in this section. Extension of these operations need to be considered in the future for handling the known partitions and partition algebra expressions attached to structural graph nodes.

DefinePartition($PartSetPath\, h[\alpha],\ PartDataset\, Rel$**)**

A valid but yet unknown partition $h[\alpha]$ is defined in-line. *Rel* is a relation (may be formed using traditional relational algebra operations from constant tables), which becomes a partition relation by this operation and will be attached to the node of $h[\alpha]$ of the structural graph.

If some of the projected partitions of $h[\alpha]$ are already known, consistency must be checked since it is an extension of the existing data.

Projections of $h[\alpha]$ become known too, as well as some other partition combinations by left-to-right join (based on functional dependencies). An algebra expression is generated and attached to the nodes of the structural graph that become known by this definition.

ImportPartition($PartSetPath\, h[\alpha],\ ExternDatasource\, ExtRel$**)**

Loads partition data from *ExtRel* into the database for $h[\alpha]$. Relation schema of *ExtRel* must match the partition schema $h[\alpha]$. If $h[\alpha]$ is already known, even partially, then consistency must be checked. The result on the logical level is the assignment of *ExtRel* to the node $h[\alpha]$.

External data source may be a relational table in an operational database or a (possibly complex) query operation. It is evaluated and its result is stored in the partition database.

Projections of $h[\alpha]$ become known too, as well as some other partition combinations by left-to-right join (based on functional dependencies). An algebra expression is generated and attached to the nodes of the structural graph that become known by this definition.

A.2 Partition Database Query Operations

As in traditional database theory, query results do not become part of the database and queries do not have 'side effects' in the database. The actual partition database is therefore considered as an implicit constant input parameter of these operations.

A.2.1 Structural Queries

These query operations are based exclusively on the structural level. They provide information about the complete partition structure and do not consider the partition data actually known.

Partition structure of a base set

GetPartAttrs($\{SetPath\}\,\mathcal{H}$**) returns** *Partition***:** Retrieves the partition and aggregation attribute names that are valid for each of the sets in $\mathcal{H}$.

GetPartStruct($SetPath\,h$**) returns** *PartStruct***:** Retrieves the valid partition structure of set h (attributes, refinements, cumulative operators). See the specific operations of *PartStruct* in Section A.3. The structure of attributes valid for the whole set h is retrieved only. Subset partitions can be queried separately.

GetPartEquivAttribs($SetPath\,h$**) returns** $\{Partition\}$**:** Partition attributes of h retrieved and grouped according to equivalence (set equality). Partition A and B of h are equivalent if and only if both constraints $A \rightarrow B$ and $B \rightarrow A$ hold. In this case A and B contain names referring to the same sets (synonyms). The synonym table can be queried as partition $h[AB]$.

Origin of partitions and subsets

GetParent($SetPath\,h$**) returns** *PartSetPath***:** Retrieves the parent partition of h according to its subset path. The operation extracts this information from h by dropping the last subset of its path (e.g. the parent of $a[B].b[C].c$ is $a[B].b[C]$). It depends on how the set is referenced and therefore, not unique for a set (e.g. if C is valid for a then $a[C].c[B].b$ refers to the same set but the parent is different from above). This operation is more useful if set aliases are used (see DeclareAlias above).

GetPrimaryBaseSet($PartSetPath\,h[A]$**) returns** *SetPath***:** If partition A is not inherited then the result is h. If it is inherited then its *root ancestor* (the attribute with the same name having the largest base set) is searched and the result is its base set, a superset of h.

GetFullExtension($SetPath\,k$**) returns** *SetPath***:** Retrieves the full extension of subset k. It is the subset with the same name in the root ancestor partition of k's parent (a subset of the primary base set).

If, for example $k = h[A].a$ and the primary base set of $h[A]$ is l then the result is $l[A].a$. See more examples in Section 6.3.6.

A.2.2 Schema Queries

These query operations take into account the structural as well as the logical level. They retrieve information about the partition structure and also reveal the unknown ways of partitioning (which are potential targets of the input of external information).

Known partitions of a base set

GetKnownPartitions($\{SetPath\}\,\mathcal{H}$**) returns** $\{Partition\}$**:** Retrieves the ways of partitioning (partition and cumulative attribute combinations) that are known for each set in $\mathcal{H}$. Actual partition data can be queried by QueryPartition (see Section A.2.3).

Syntax may be extended to accept *PartSetPath* values. In that case, a known partition schema is retrieved only if it contains the schema specified in the *PartSetPath* value(s).

GetKnownPartStruct(*SetPath h***) returns** *HyPartStruct***:** Retrieves the known partition structure of h, which is part of the full structural graph retrieved by GetPartStruct. It contains each node that can be queried, i.e. has attached partition algebra expression on the logical level. The graph can be partial, i.e. it may be cut and have more maximal nodes as in Section 5.2.1.

GetPartExpr(*PartSetPath* $h[\alpha]$**) returns** *PartAlgExpr***:** Retrieves the attached algebra expression or UNKNOWN if no expression is attached to the node of $h[\alpha]$ (if $h[\alpha]$ is not a closed set then a projection of the expression attached to its closure is taken).

Navigation operations

The following operations give examples of navigation as in a data cube. They are based on structural graph operations (see Section A.3). These are schema queries, actual data can be retrieved by QueryPartition (see Section A.2.3).

If explicit dimensions are needed, extension of the syntax with namespaces can be introduced.

GetRollUps(*PartSetPath* $h[\alpha]$**) returns** $\{$*Partition*$\}$**:** Queries possible roll-up schemata of $h[\alpha]$ using GetPSCoarserPartitions. A roll-up always omits some of the partition attributes and may extend the schema with attributes defining a one-step coarser partition of the same set. The user must choose one of the roll-ups and query its data using QueryPartition (see below).

GetDirectRefinements(*PartSetPath* $h[\alpha]$**) returns** $\{$*Partition*$\}$**:** Queries one-step refinement schemata of $h[\alpha]$ using GetPSFinerPartitions. It keeps all attributes of the schema and extends them with an attribute (or more, equivalent attributes) defining a finer partition of the same set.

GetDrillDowns(*PartSetPath* $h[\alpha]$**) returns** $\{$*Partition*$\}$**:** Using GetPSPrunedRefinements, this operation queries refinements as GetDirectRefinements but omits attributes becoming redundant (attributes referring to partitions being refined).

GetMaxEquiv(*PartSetPath* $h[\alpha]$**) returns** *Partition***:** Retrieves the maximal equivalent partition schema. It is the partition according to α^+ (closure of α) if known. If not, the largest known superset is taken. The resulting partition is equivalent with $h[\alpha]$ (refers to the same subsets and the same node in the structural graph).

GetMinEquiv(*PartSetPath* $h[\alpha]$**) returns** $\{$*Partition*$\}$**:** Retrieves minimal equivalent partition schemata. Partitions according to minimal attribute sets are retrieved whose closure contains α. Each resulting partition is equivalent with $h[\alpha]$ and is a *key* of the attribute set α.

Querying and navigating subset partitions

Each of the following operations take a PartSetPath type parameter but a SetPath value can also be accepted as a base set without partitions. In such a case all valid partition attributes of the specified base set is taken as default.

GetPartitionedSubsets(*PartSetPath* $h[\alpha]$**) returns** $\{$*SetPath*$\}$**:** Retrieves each subset of h defined by an attribute of α which has own partition or cumulative attributes whose base set is a real superset of h. For such a partition the subset must be selected in order to drill down according to that attribute.

GetKnownSubPartitions($\mathit{PartSetPath}\, h[\alpha]$**) returns** $\{\mathit{PartSetPath}\}$**:** Retrieves the finest known partitions of sets queried by GetPartitionedSubsets. Partitions valid for the whole h as well (such as inherited partitions) are not included in the result.

GetDirectSubRefinements($\mathit{PartSetPath}\, h[\alpha]$**) returns** $\{\mathit{PartSetPath}\}$**:** Retrieves one-step refinements that only valid (and known) for real subsets of h. For each set k in the result of GetPartitionedSubsets, direct refinements of $k[\alpha]$ (not valid for the whole set h) are taken.

A.2.3 Querying Partition Data

Queries of this section are based on the schema (structural and logical level) as well as on the actual data (instance level).

Direct Relation Query

QueryPartition($\mathit{PartSetPath}\, h[\alpha]$**) returns** *PartDataset***:** Corresponds to a SELECT-FROM query. Queries partition α of h. If it is known, the attached algebra expression is evaluated and the resulting partition relation is retrieved. If α is not a closed attribute set then the appropriate projection of the expression attached to its closure is taken.

If each attribute of α is valid for h but the partition according to their combination is not known, the result will be the special constant UNKNOWN. If one or more of the attributes is not valid then the operation fails.

QueryPartition($\mathit{PartSetPath}\, h[\alpha]$, $\mathit{Condition}\, \theta$**) returns** *PartDataset***:** This extended syntax corresponds to a simple SELECT-FROM-WHERE query. θ is a subset selection condition on the attributes of h. The partition according to the attributes of θ must be known together with α.

Search

GetCommonSuperset($\{\mathit{SetPath}\}\, \mathcal{H}$**) returns** *SetPath***:** Looks for a common superset of sets $\mathcal{H}$. It is determined by the set path expressions and is therefore not always the smallest common superset that can be expressed by the known data.

SearchSubsetsKnown($\mathit{SetPath}\, k$, $\mathit{SetSymb}\, a$**) returns** $\{\mathit{SetPath}\}$**:** Looks for sets named a among the subsets of k in any of the partitions. Due to unknown partitions, containment of some sets may not be decidable. Ambiguous sets are not returned.

SearchSupersetsKnown($\mathit{SetPath}\, k$, $\mathit{SetSymb}\, a$**) returns** $\{\mathit{SetPath}\}$**:** Looks for sets named a among the supersets of k, similarly to SearchSubsetsKnown.

Set Relationships

Relationships of two sets can be queried by the following operations. The answer can be YES, NO or UNKNOWN. If the two sets are not elements of partitions of the same base set, a common superset is taken as the basis of comparison. Inheritance and characteristic partition may be used implicitly for the comparison.

A partition database represents nonempty sets. It must be checked whether a set path corresponds to an empty set, using IsEmpty. Set relationship operations are valid only if both sets are nonempty.

IsEmpty($SetPath\, h$**) returns** *Logical***:** $h = \emptyset$?

IsPartOf($SetPath\, h$, $SetPath\, k$**) returns** *Logical***:** $h \subseteq i$?

IsDisjoint($SetPath\, h$, $SetPath\, k$**) returns** *Logical***:** $h \cap i = \emptyset$?

IsEqual($SetPath\, h$, $SetPath\, k$**) returns** *Logical***:** $h = i$?

GetRelationship($SetPath\, h$, $SetPath\, k$**) returns** $(Logical, Logical, Logical)$**:** A general query regarding the relationship of h and k. The first component of the result represents whether $h \setminus k$ is nonempty, the second refers to $h \cap k$ and the third refers to $k \setminus h$. The above operations are special cases of this.

Set relationship may be definite or indefinite (missing information). Special constants for triples may be defined that give names to relationships. A definite relationship can be one of the following (assuming both sets are nonempty) cases:

- (NO,YES,NO)=setEqual,
- (NO,YES,YES)=setSubset, (YES,YES,NO)=setSuperset,
- (YES,NO,YES)=setDisjoint,
- (YES,YES,YES)=setOverlap

An indefinite relationship may be one of the following (suffix '*Ov*' means 'or overlap', similarly to the conventional '*Eq*' meaning 'or equal'):

- (NO,YES,UNKNOWN)=setSubsetEq, (UNKNOWN,YES,NO)=setSupersetEq,
- (YES,UNKNOWN,UNKNOWN)=setNoSubsetEq,
  (UNKNOWN,UNKNOWN,YES)=setNoSupersetEq,
- (UNKNOWN,YES,YES)=setSubsetOv, (YES,YES,UNKNOWN)=setSupersetOv,
- (YES,UNKNOWN,YES)=setNoSubsetSuperset=setDisjointOv,
- (UNKNOWN,YES,UNKNOWN)=setNonDisjoint,
- (UNKNOWN,UNKNOWN,UNKNOWN)=setUnknown

Other value combinations are either contradictory (imply the emptiness of one of the sets) or imply one of the listed combinations (assuming both sets are nonempty, an UNKNOWN value must be replaced by YES).

It is an open question whether each of the indefinite relationships can occur in a partition database.

A.3 Structural Graph Operations

PartStruct values are used for partition declaration for a base set on the structural level (see DeclarePartiton). The sub-structure of known partitions can be obtained as a HyPartStruct value. Partition structure can be represented by a structural graph. The following operations manipulate such graphs and can be used for structure construction and navigation queries.

Partition database is not considered as implicit parameter of these operations since they operate on structural graphs that can be obtained from or passed to the database by other operations discussed above.

A.3.1 Structural Graph Construction

PartStructCreate($\mathit{Partition}\,[\alpha]$, $\{\mathit{FunctDep}\}\,\mathcal{F}$, $\{\mathit{AggrCumulOp}\}\,\mathcal{A}$**)** *returns PartStruct***:** Constructs a partition structure from the given data. The semilattice of different partitions is created with attributes $[\alpha]$ and refinement declarations $\mathcal{F}$ (functional dependencies). Cumulative attributes $\mathcal{A}$ are added globally to the graph (as a future extension, local assignment shall be considered node-by-node).

PartStructProduct($\mathit{PartStruct}\,\mathcal{G}$, $\mathit{PartStruct}\,\mathcal{H}$**)** *returns PartStruct***:** Constructs the direct product of two partition structures. $\mathcal{G}$ and $\mathcal{H}$ cannot contain partition attributes with the same name. If $\mathcal{G}$ and $\mathcal{H}$ correspond to two different dimension structures of a multidimensional schema, the two-dimensional cube schema is constructed by this operation. Cumulative attributes are taken from both structures. The syntax may be extended in order to accept more than two parameters.

PartStructNest($\mathit{PartStruct}\,\mathcal{G}$, $\mathit{PartStruct}\,\mathcal{H}$**)** *returns PartStruct***:** Constructs the direct sum of two partition structures, i.e. $\mathcal{H}$ is nested into $\mathcal{G}$. Partition by each closed attribute set of $\mathcal{H}$ will be a refinement of all partitions by closed sets of $\mathcal{G}$.

PartStructAddCumulAttr($\mathit{PartStruct}\,\mathcal{G}$, $\{\mathit{AggrCumulOp}\}\,\mathcal{A}$**):** adds new cumulative attributes to the graph. The syntax can be extended to allow cumulative attributes attached to specific nodes.

PartStructAddClosed($\mathit{PartStruct}\,\mathcal{G}$, $\mathit{Partition}\,[\alpha]$**)** *returns PartStruct* **and**
PartStructDelClosed($\mathit{PartStruct}\,\mathcal{G}$, $\mathit{Partition}\,[\alpha]$**)** *returns PartStruct***:** Addition or deletion of a closed attribute set. Adding a set invalidates some of the refinement dependencies. Deletion implies new dependencies.

PartStructAddConstr($\mathit{PartStruct}\,\mathcal{G}$, $\{\mathit{FunctDep}\}\,\mathcal{F}$**)** *returns PartStruct* **and**
PartStructDelConstr($\mathit{PartStruct}\,\mathcal{G}$, $\{\mathit{FunctDep}\}\,\mathcal{F}$**)** *returns PartStruct***:** Adding or deleting functional dependencies (refinement constraints). An added constraint may imply others. Deleting is only valid if the deleted constraints are not implied by the remaining set.

Further construction operations may be introduced to reach completeness wrt the four-level partition database framework, such as attribute addition, deletion, rename, unification of two graphs with common attributes, etc.

A.3.2 Structural Graph Queries

The following operations can be used to get information of and navigate along the structural graph of the partition database. These are schema queries, the actual data can be queried from the database by QueryPartition (see Section A.2.3.

GetPartStruct retrieves the graph for a base set, based on the structural level. It is always a full graph (PartStruct value). GetKnownPartStruct retrieves the structure of known partitions (HyPartStruct value). PartStruct may be treated as a subtype of HyPartStruct, therefore the following operations can be used for both PartStruct and HyPartStruct values.

GetPSNodes($\mathit{HyPartStruct}\,\mathcal{G}$**) returns** $\{\mathit{Partition}\}$**:** Retrieves the nodes of $\mathcal{G}$, i.e. the schemata of different (non-equivalent) partitions.

GetPSMaxNodes($\mathit{HyPartStruct}\,\mathcal{G}$**) returns** *HyPartition***:** Retrieves the finest partition(s). A PartStruct value has always one maximal node.

GetPSPartAttrs($\mathit{HyPartStruct}\,\mathcal{G}$**) returns** $\{\mathit{PartSymb}\}$**:** Retrieves the partition attribute names.

GetPSAggregAttrs($\mathit{HyPartStruct}\,\mathcal{G}$**) returns** $\{\mathit{AggrCumulOp}\}$**:** Retrieves the cumulative attributes with their default operators (as a future extension, node-by-node validity of aggregation attributes will be needed).

GetPSFunctDeps($\mathit{HyPartStruct}\,\mathcal{G}$**) returns** $\{\mathit{FunctDep}\}$**:** Retrieves the set of functional dependencies (refinement constraints). A closed set of functional dependencies may be considered as another data type in the future with own operations (eg. the task of retrieving whether a particular constraint is an element of the set or not may be performed by on-demand derivation).

GetPSRelatedPartitions($\mathit{HyPartStruct}\,\mathcal{G}$, $\mathit{Partition}\,[\alpha]$**) returns** $\{\mathit{Partition}\}$**:** Retrieves the nodes that are not independent of attributes α. Two attribute sets are treated as dependent if a functional dependency exists from one to another.

GetPSMaxEquiv($\mathit{HyPartStruct}\,\mathcal{G}$, $\mathit{Partition}\,[\alpha]$**:** Retrieves maximal equivalent partition schema with α. It is the node corresponding to α (more precisely, α^+).

GetMinEquiv($\mathit{PartSetPath}\,h[\alpha]$**) returns** $\{\mathit{Partition}\}$**:** Retrieves minimal equivalent partition schemata with α by removing redundant attributes in each possible way. An attribute is redundant if omitting it yields the same (an equivalent) partition, therefore, each attribute set (partition schema) is the result corresponds to the same node in the structural graph as α. The results are the keys of the attribute set α.

GetPSCoarserPartitions($\mathit{HyPartStruct}\,\mathcal{G}$, $\mathit{Partition}\,[\alpha]$**) returns** $\{\mathit{Partition}\}$**:** Retrieves the one-step-coarsements (roll-up schemata) of partition α. For each possible way, it removes one attribute (or more, equivalent attributes) but may add attributes which the removed attribute is non-transitively dependent on. It simply corresponds to a step towards the minimal element (closure of the empty set) of the structural (semilattice) graph.

GetPSFinerPartitions($\mathit{HyPartStruct}\,\mathcal{G}$, $\mathit{Partition}\,[\alpha]$**) returns** $\{\mathit{Partition}\}$**:** Retrieves the one-step-refinements (drill-down schemata) of partition α. It always adds one (or more, equivalent) attribute to the schema each possible way, such that at least one attribute in α is non-transitively dependent on it. It simply corresponds to a step towards a maximal element of the semilattice graph.

GetPSPrunedRefinements($\mathit{HyPartStruct}\,\mathcal{G}$, $\mathit{Partition}\,[\alpha]$**) returns** $\{\mathit{Partition}\}$**:** It works as GetPS-FinerPartitions but removes the attributes became redundant, i.e. those attributes that functionally dependent on the new attributes included during refinement (and were not dependent on the attributes of α before refinement).

Appendix B

Sets of Functional Dependencies in Spreadsheet Form

B.1 The Quaternary Case

Chapter 8 introduced the spreadsheet notation for sets of functional dependencies.

All sets of functional dependencies for 4 attributes are presented here in the tabular form, grouped by value combinations of attribute dimensions (Table B.1). Sets equivalent up to permutation of attributes are treated as one single case with a representant set presented (the total number of sets is 2 271, treating equivalent sets as one case we get these 165 cases, see also Section 8.2.3). Due to space limitations, dimension values of attributes are written in separate rows (class headers) and the binary representation forms a single column. Grouped bits represent the presence or absence of dependencies in the following order (from left to right):

$(BCD \to A,\ ACD \to B,\ ABD \to C,\ ABC \to D)$,
$(BC \to A,\ AC \to B,\ AB \to C)$, $(BD \to A,\ AD \to B,\ AB \to D)$,
$(CD \to A,\ AD \to C,\ AC \to D)$, $(CD \to B,\ BD \to C,\ BC \to D)$,
$(B \to A,\ A \to B)$, $(C \to A,\ A \to C)$, $(D \to A,\ A \to D)$,
$(C \to B,\ B \to C)$, $(D \to B,\ B \to D)$, $(D \to C,\ C \to D)$.

Contradictory classes are also indicated in the table as 'no valid sets'.

#	Binary representation	Generating system of functional dependencies
	$[A]=\infty, [B]=\infty, [C]=\infty, [D]=\infty$	
0	0000 000 000 000 000 00 00 00 00 00 00	$\emptyset$
	$[A]=1, [B]=\infty, [C]=\infty, [D]=\infty$	
1	1000 100 100 000 000 10 00 00 00 00 00	$\{B \to A\}$
2	1000 100 100 100 000 10 00 00 00 00 00	$\{CD \to A, B \to A\}$
3	1000 100 100 100 000 10 10 00 00 00 00	$\{B \to A, C \to A\}$
4	1000 100 100 100 000 10 10 10 00 00 00	$\{B \to A, C \to A, D \to A\}$
	$[A]=1, [B]=1, [C]=\infty, [D]=\infty$	
5	1100 100 010 100 100 00 10 00 00 10 00	$\{C \to A, D \to B\}$
6	1100 110 000 100 100 00 10 00 10 00 00	$\{C \to B, C \to A\}$

#	Binary representation	Generating system of functional dependencies
7	1100 110 100 100 100 00 10 00 10 00 00	$\{BD \to A, C \to B, C \to A\}$
8	1100 110 110 100 100 00 10 00 10 00 00	$\{BD \to A, AD \to B, C \to B, C \to A\}$
9	1100 110 100 100 100 00 10 10 10 00 00	$\{C \to B, C \to A, D \to A\}$
10	1100 110 110 100 100 00 10 10 10 10 00	$\{D \to B, C \to B, C \to A, D \to A\}$
11	1100 110 100 100 100 10 10 00 10 00 00	$\{C \to B, B \to A\}$
12	1100 110 110 100 100 10 10 00 10 00 00	$\{AD \to B, C \to B, B \to A\}$
13	1100 110 100 100 100 10 10 10 10 00 00	$\{C \to B, B \to A, D \to A\}$
14	1100 110 110 100 100 10 10 10 10 10 00	$\{D \to B, C \to B, B \to A\}$
15	1100 110 110 000 000 11 00 00 00 00 00	$\{A \to B, B \to A\}$
16	1100 110 110 100 100 11 00 00 00 00 00	$\{CD \to B, A \to B, B \to A\}$
17	1100 110 110 100 100 11 10 00 10 00 00	$\{C \to B, A \to B, B \to A\}$
18	1100 110 110 100 100 11 10 10 10 10 00	$\{D \to B, C \to B, A \to B, B \to A\}$
$[A] = 1, [B] = 1, [C] = 1, [D] = \infty$		
19	1110 000 110 110 110 00 00 10 00 10 10	$\{D \to A, D \to B, D \to C\}$
20	1110 100 110 110 110 00 00 10 00 10 10	$\{BC \to A, D \to B, D \to C\}$
21	1110 110 110 110 110 00 00 10 00 10 10	$\{BC \to A, AC \to B, D \to B, D \to C\}$
22	1110 111 110 110 110 00 00 10 00 10 10	$\{BC \to A, AC \to B, AB \to C, D \to B, D \to C\}$
23	1110 100 110 110 110 10 00 10 00 10 10	$\{D \to B, B \to A, D \to C\}$
24	1110 110 110 110 110 10 00 10 00 10 10	$\{AC \to B, D \to B, B \to A, D \to C\}$
25	1110 101 110 110 110 10 00 10 01 10 10	$\{D \to B, B \to C, B \to A\}$
26	1110 111 110 110 110 10 00 10 01 10 10	$\{AC \to B, D \to B, B \to C, B \to A\}$
27	1110 100 110 110 110 10 10 10 00 10 10	$\{D \to B, B \to A, C \to A, D \to C\}$
28	1110 110 110 110 110 10 10 10 10 10 10	$\{C \to B, B \to A, D \to C\}$
29	1110 110 110 010 010 11 00 00 00 00 10	$\{A \to B, B \to A, D \to C\}$
30	1110 110 110 110 110 11 00 10 00 10 10	$\{D \to B, A \to B, B \to A, D \to C\}$
31	1110 111 110 010 010 11 01 00 01 00 00	$\{A \to C, A \to B, B \to A\}$
32	1110 111 110 110 110 11 01 00 01 00 00	$\{CD \to B, A \to C, A \to B, B \to A\}$
33	1110 111 110 010 010 11 01 00 01 00 10	$\{A \to C, A \to B, B \to A, D \to C\}$
34	1110 111 110 110 110 11 01 10 01 10 10	$\{D \to B, A \to C, A \to B, B \to A\}$
35	1110 110 110 110 110 11 10 10 10 10 10	$\{C \to B, A \to B, B \to A, D \to C\}$
36	1110 111 110 110 110 11 11 00 11 00 00	$\{A \to C, A \to B, B \to A, C \to A\}$
37	1110 111 110 110 110 11 11 10 11 10 10	$\{A \to C, A \to B, B \to A, C \to A, D \to C\}$
$[A] = 1, [B] = 1, [C] = 1, [D] = 1$		
38	1111 110 110 011 011 11 00 00 00 00 11	$\{A \to B, B \to A, C \to D, D \to C\}$
39	1111 111 111 011 011 11 01 01 01 01 00	$\{B \to A, A \to C, A \to B, A \to D\}$
40	1111 111 111 111 111 11 01 01 01 01 00	$\{CD \to B, B \to A, A \to C, A \to D\}$
41	1111 111 111 011 011 11 01 01 01 01 10	$\{B \to A, A \to B, A \to D, D \to C\}$
42	1111 110 110 111 111 11 10 10 10 10 11	$\{C \to A, A \to B, B \to A, C \to D, D \to C\}$
43	1111 111 111 111 111 11 11 01 11 01 01	$\{C \to B, B \to A, A \to C, A \to D\}$
44	1111 111 111 111 111 11 11 11 11 11 11	$\{C \to B, B \to A, A \to D, D \to C\}$
$[A] = 1, [B] = 1, [C] = 1, [D] = 2$		
45	1111 100 110 110 111 00 00 10 00 10 10	$\{BC \to D, BC \to A, D \to B, D \to C\}$
46	1111 110 110 111 111 00 00 10 00 10 10	$\{BC \to D, BC \to A, AC \to B, D \to B, D \to C\}$

#	Binary representation	Generating system of functional dependencies
47	1111 111 111 111 111 00 00 10 00 10 10	$\{BC \to D, BC \to A, AC \to B, AB \to C, D \to B, D \to C\}$
48	1111 100 110 110 111 10 00 10 00 10 10	$\{BC \to D, D \to B, B \to A, D \to C\}$
49	1111 110 110 111 111 10 00 10 00 10 10	$\{BC \to D, AC \to B, D \to B, B \to A, D \to C\}$
50	1111 100 110 110 111 10 10 10 00 10 10	$\{BC \to D, D \to B, B \to A, C \to A, D \to C\}$
51	1111 110 110 011 011 11 00 00 00 00 10	$\{BC \to D, A \to B, B \to A, D \to C\}$
52	1111 110 110 111 111 11 00 10 00 10 10	$\{BC \to D, D \to B, A \to B, B \to A, D \to C\}$
$[A] = 1, [B] = 1, [C] = 1, [D] = 3$		
53	1111 000 110 110 110 00 00 10 00 10 10	$\{ABC \to D, D \to A, D \to B, D \to C\}$
$[A] = 1, [B] = 1, [C] = 2, [D] = \infty$		
54	1110 100 010 110 100 00 10 00 00 10 00	$\{AD \to C, C \to A, D \to B\}$
55	1110 101 010 110 100 00 10 00 00 10 00	$\{AB \to C, C \to A, D \to B\}$
56	1110 110 100 100 110 00 10 00 10 00 00	$\{BD \to C, C \to B, C \to A\}$
57	1110 110 110 110 110 00 10 00 10 00 00	$\{BD \to C, AD \to B, C \to B, C \to A\}$
58	1110 110 100 100 110 00 10 10 10 00 00	$\{BD \to C, C \to B, C \to A, D \to A\}$
59	1110 111 000 100 100 00 10 00 10 00 00	$\{AB \to C, C \to B, C \to A\}$
60	1110 111 100 100 110 00 10 00 10 00 00	$\{BD \to C, AB \to C, C \to B, C \to A\}$
61	1110 111 110 110 110 00 10 00 10 00 00	$\{BD \to C, AB \to C, AD \to B, C \to B, C \to A\}$
62	1110 111 100 100 110 00 10 10 10 00 00	$\{AB \to C, C \to B, C \to A, D \to A\}$
63	1110 110 100 100 110 10 10 00 10 00 00	$\{BD \to C, C \to B, B \to A\}$
64	1110 110 110 110 110 10 10 00 10 00 00	$\{BD \to C, AD \to B, C \to B, B \to A\}$
65	1110 110 100 100 110 10 10 10 10 00 00	$\{BD \to C, C \to B, B \to A, D \to A\}$
66	1110 110 110 010 010 11 00 00 00 00 00	$\{BD \to C, A \to B, B \to A\}$
67	1110 110 110 110 110 11 00 00 00 00 00	$\{CD \to B, BD \to C, A \to B, B \to A\}$
68	1110 110 110 110 110 11 10 00 10 00 00	$\{BD \to C, C \to B, A \to B, B \to A\}$
$[A] = 1, [B] = 1, [C] = 2, [D] = 2$		
69	1111 100 010 110 101 00 10 00 00 10 00	$\{AD \to C, BC \to D, C \to A, D \to B\}$
70	1111 101 011 110 101 00 10 00 00 10 00	$\{BC \to D, AB \to C, C \to A, D \to B\}$
71	1111 110 110 011 011 11 00 00 00 00 00	$\{BD \to C, BC \to D, A \to B, B \to A\}$
72	1111 110 110 111 111 11 00 00 00 00 00	$\{CD \to B, BD \to C, BC \to D, A \to B, B \to A\}$
$[A] = 1, [B] = 1, [C] = 2, [D] = 3$		
no valid sets		
$[A] = 1, [B] = 1, [C] = 3, [D] = \infty$		
73	1110 110 000 100 100 00 10 00 10 00 00	$\{ABD \to C, C \to B, C \to A\}$
$[A] = 1, [B] = 1, [C] = 3, [D] = 3$		
no valid sets		
$[A] = 1, [B] = 2, [C] = \infty, [D] = \infty$		
74	1100 100 000 100 100 00 10 00 00 00 00	$\{CD \to B, C \to A\}$
75	1100 100 010 100 100 00 10 00 00 00 00	$\{AD \to B, C \to A\}$
76	1100 100 100 100 100 00 10 00 00 00 00	$\{CD \to B, BD \to A, C \to A\}$
77	1100 100 110 100 100 00 10 00 00 00 00	$\{BD \to A, AD \to B, C \to A\}$
78	1100 100 100 100 100 00 10 10 00 00 00	$\{CD \to B, C \to A, D \to A\}$
79	1100 100 100 100 100 10 00 00 00 00 00	$\{CD \to B, B \to A\}$

#	Binary representation	Generating system of functional dependencies
80	1100 110 100 000 000 10 00 00 00 00 00	$\{AC \to B, B \to A\}$
81	1100 110 100 100 100 10 00 00 00 00 00	$\{CD \to B, AC \to B, B \to A\}$
82	1100 110 110 000 000 10 00 00 00 00 00	$\{AC \to B, AD \to B, B \to A\}$
83	1100 110 110 100 100 10 00 00 00 00 00	$\{CD \to B, AC \to B, AD \to B, B \to A\}$
84	1100 100 100 100 100 10 10 00 00 00 00	$\{CD \to B, B \to A, C \to A\}$
85	1100 100 110 100 100 10 10 00 00 00 00	$\{AD \to B, B \to A, C \to A\}$
86	1100 100 100 100 100 10 10 10 00 00 00	$\{CD \to B, B \to A, C \to A, D \to A\}$
$[A] = 1, [B] = 2, [C] = 2, [D] = \infty$		
87	1110 000 100 100 110 00 00 10 00 00 00	$\{CD \to B, BD \to C, D \to A\}$
88	1110 010 100 100 110 00 00 10 00 00 00	$\{BD \to C, AC \to B, D \to A\}$
89	1110 011 100 100 110 00 00 10 00 00 00	$\{AC \to B, AB \to C, D \to A\}$
90	1110 100 100 100 110 00 00 10 00 00 00	$\{CD \to B, BD \to C, BC \to A, D \to A\}$
91	1110 110 100 100 110 00 00 10 00 00 00	$\{BD \to C, BC \to A, AC \to B, D \to A\}$
92	1110 111 100 100 110 00 00 10 00 00 00	$\{BC \to A, AC \to B, AB \to C, D \to A\}$
93	1110 100 100 100 110 10 00 00 00 00 00	$\{CD \to B, BD \to C, B \to A\}$
94	1110 100 110 010 010 10 00 00 00 00 00	$\{BD \to C, AD \to B, B \to A\}$
95	1110 100 110 110 110 10 00 00 00 00 00	$\{CD \to B, BD \to C, AD \to B, B \to A\}$
96	1110 100 100 100 110 10 00 10 00 00 00	$\{CD \to B, BD \to C, B \to A, D \to A\}$
97	1110 110 100 000 010 10 00 00 00 00 00	$\{BD \to C, AC \to B, B \to A\}$
98	1110 110 100 100 110 10 00 00 00 00 00	$\{CD \to B, BD \to C, AC \to B, B \to A\}$
99	1110 110 110 010 010 10 00 00 00 00 00	$\{BD \to C, AC \to B, AD \to B, B \to A\}$
100	1110 110 110 110 110 10 00 00 00 00 00	$\{CD \to B, BD \to C, AC \to B, AD \to B, B \to A\}$
101	1110 110 100 100 110 10 00 10 00 00 00	$\{BD \to C, AC \to B, B \to A, D \to A\}$
102	1110 100 100 100 110 10 10 00 00 00 00	$\{CD \to B, BD \to C, B \to A, C \to A\}$
103	1110 100 110 110 110 10 10 00 00 00 00	$\{BD \to C, AD \to B, B \to A, C \to A\}$
104	1110 100 100 100 110 10 10 10 00 00 00	$\{CD \to B, BD \to C, B \to A, C \to A, D \to A\}$
$[A] = 1, [B] = 2, [C] = 2, [D] = 2$		
105	1111 100 100 100 111 10 00 00 00 00 00	$\{CD \to B, BD \to C, BC \to D, B \to A\}$
106	1111 110 100 001 011 10 00 00 00 00 00	$\{BD \to C, BC \to D, AC \to B, B \to A\}$
107	1111 110 100 101 111 10 00 00 00 00 00	$\{CD \to B, BD \to C, BC \to D, AC \to B, B \to A\}$
108	1111 110 110 011 011 10 00 00 00 00 00	$\{BD \to C, BC \to D, AC \to B, AD \to B, B \to A\}$
109	1111 110 110 111 111 10 00 00 00 00 00	$\{CD \to B, BD \to C, BC \to D, AC \to B, AD \to B, B \to A\}$
110	1111 100 100 100 111 10 10 00 00 00 00	$\{CD \to B, BD \to C, BC \to D, B \to A, C \to A\}$
111	1111 100 110 110 111 10 10 00 00 00 00	$\{BD \to C, BC \to D, AD \to B, B \to A, C \to A\}$
112	1111 100 100 100 111 10 10 10 00 00 00	$\{CD \to B, BD \to C, BC \to D, B \to A, C \to A, D \to A\}$
$[A] = 1, [B] = 2, [C] = 2, [D] = 3$		
113	1111 000 100 100 110 00 00 10 00 00 00	$\{ABC \to D, CD \to B, BD \to C, D \to A\}$
$[A] = 1, [B] = 2, [C] = 3, [D] = \infty$		
114	1110 100 000 100 100 00 10 00 00 00 00	$\{ABD \to C, CD \to B, C \to A\}$
$[A] = 1, [B] = 2, [C] = 3, [D] = 3$		
no valid sets		
$[A] = 1, [B] = 3, [C] = \infty, [D] = \infty$		

#	Binary representation	Generating system of functional dependencies
115	1100 100 100 000 000 10 00 00 00 00 00	$\{ACD \to B, B \to A\}$
	$[A] = 1, [B] = 3, [C] = 3, [D] = \infty$	
	no valid sets	
	$[A] = 1, [B] = 3, [C] = 3, [D] = 3$	
	no valid sets	
	$[A] = 2, [B] = \infty, [C] = \infty, [D] = \infty$	
116	1000 100 000 000 000 00 00 00 00 00 00	$\{BC \to A\}$
117	1000 100 100 000 000 00 00 00 00 00 00	$\{BC \to A, BD \to A\}$
118	1000 100 100 100 000 00 00 00 00 00 00	$\{BC \to A, BD \to A, CD \to A\}$
	$[A] = 2, [B] = 2, [C] = \infty, [D] = \infty$	
119	1100 000 000 100 100 00 00 00 00 00 00	$\{CD \to B, CD \to A\}$
120	1100 100 000 100 100 00 00 00 00 00 00	$\{CD \to B, BC \to A\}$
121	1100 100 010 000 000 00 00 00 00 00 00	$\{BC \to A, AD \to B\}$
122	1100 100 010 100 100 00 00 00 00 00 00	$\{CD \to B, BC \to A, AD \to B\}$
123	1100 100 100 100 100 00 00 00 00 00 00	$\{CD \to B, BC \to A, BD \to A\}$
124	1100 110 000 000 000 00 00 00 00 00 00	$\{BC \to A, AC \to B\}$
125	1100 110 000 100 100 00 00 00 00 00 00	$\{CD \to B, BC \to A, AC \to B\}$
126	1100 110 100 000 000 00 00 00 00 00 00	$\{BC \to A, AC \to B, BD \to A\}$
127	1100 110 100 100 100 00 00 00 00 00 00	$\{CD \to B, BC \to A, AC \to B, BD \to A\}$
128	1100 110 110 000 000 00 00 00 00 00 00	$\{BC \to A, AC \to B, BD \to A, AD \to B\}$
129	1100 110 110 100 100 00 00 00 00 00 00	$\{CD \to B, BC \to A, AC \to B, BD \to A, AD \to B\}$
	$[A] = 2, [B] = 2, [C] = 2, [D] = \infty$	
130	1110 000 110 010 010 00 00 00 00 00 00	$\{BD \to C, BD \to A, AD \to B\}$
131	1110 000 110 110 110 00 00 00 00 00 00	$\{CD \to B, BD \to C, BD \to A, AD \to B\}$
132	1110 100 010 010 000 00 00 00 00 00 00	$\{AD \to C, BC \to A, AD \to B\}$
133	1110 100 100 100 110 00 00 00 00 00 00	$\{CD \to B, BD \to C, BC \to A\}$
134	1110 100 110 010 010 00 00 00 00 00 00	$\{BD \to C, BC \to A, AD \to B\}$
135	1110 100 110 110 110 00 00 00 00 00 00	$\{CD \to B, BD \to C, BC \to A, AD \to B\}$
136	1110 110 100 000 010 00 00 00 00 00 00	$\{BD \to C, BC \to A, AC \to B\}$
137	1110 110 100 100 110 00 00 00 00 00 00	$\{CD \to B, BD \to C, BC \to A, AC \to B\}$
138	1110 110 110 010 010 00 00 00 00 00 00	$\{BD \to C, BC \to A, AC \to B, AD \to B\}$
139	1110 110 110 110 110 00 00 00 00 00 00	$\{CD \to B, BD \to C, BC \to A, AC \to B, AD \to B\}$
140	1110 111 000 000 000 00 00 00 00 00 00	$\{BC \to A, AC \to B, AB \to C\}$
141	1110 111 100 000 010 00 00 00 00 00 00	$\{BD \to C, BC \to A, AC \to B, AB \to C\}$
142	1110 111 110 010 010 00 00 00 00 00 00	$\{BD \to C, BC \to A, AC \to B, AB \to C, AD \to B\}$
143	1110 111 110 110 110 00 00 00 00 00 00	$\{CD \to B, BD \to C, BC \to A, AC \to B, AB \to C, AD \to B\}$
	$[A] = 2, [B] = 2, [C] = 2, [D] = 2$	
144	1111 100 010 010 001 00 00 00 00 00 00	$\{AD \to C, BC \to D, BC \to A, AD \to B\}$
145	1111 110 100 001 011 00 00 00 00 00 00	$\{BD \to C, BC \to D, BC \to A, AC \to B\}$
146	1111 110 110 011 011 00 00 00 00 00 00	$\{BD \to C, BC \to D, BC \to A, AC \to B, AD \to B\}$
147	1111 111 001 001 001 00 00 00 00 00 00	$\{BC \to D, BC \to A, AC \to B, AB \to C\}$
148	1111 111 101 001 011 00 00 00 00 00 00	$\{BD \to C, BC \to D, BC \to A, AC \to B, AB \to C\}$

#	Binary representation	Generating system of functional dependencies
149	1111 111 111 011 011 00 00 00 00 00 00	$\{BD \to C, BC \to D, BC \to A, AC \to B, AB \to C, AD \to B\}$
150	1111 111 111 111 111 00 00 00 00 00 00	$\{CD \to B, BD \to C, BC \to D, BC \to A,$ $AC \to B, AB \to C, AD \to B\}$
$[A] = 2, [B] = 2, [C] = 2, [D] = 3$		
151	1111 000 110 010 010 00 00 00 00 00 00	$\{ABC \to D, BD \to C, BD \to A, AD \to B\}$
152	1111 000 110 110 110 00 00 00 00 00 00	$\{ABC \to D, CD \to B, BD \to C, BD \to A, AD \to B\}$
$[A] = 2, [B] = 2, [C] = 3, [D] = \infty$		
153	1110 000 000 100 100 00 00 00 00 00 00	$\{ABD \to C, CD \to B, CD \to A\}$
154	1110 100 000 100 100 00 00 00 00 00 00	$\{ABD \to C, CD \to B, BC \to A\}$
155	1110 110 000 000 000 00 00 00 00 00 00	$\{ABD \to C, BC \to A, AC \to B\}$
156	1110 110 000 100 100 00 00 00 00 00 00	$\{ABD \to C, CD \to B, BC \to A, AC \to B\}$
$[A] = 2, [B] = 2, [C] = 3, [D] = 3$		
157	1111 000 000 100 100 00 00 00 00 00 00	$\{ABD \to C, ABC \to D, CD \to B, CD \to A\}$
$[A] = 2, [B] = 3, [C] = \infty, [D] = \infty$		
158	1100 100 000 000 000 00 00 00 00 00 00	$\{ACD \to B, BC \to A\}$
159	1100 100 100 000 000 00 00 00 00 00 00	$\{ACD \to B, BC \to A, BD \to A\}$
$[A] = 2, [B] = 3, [C] = 3, [D] = \infty$		
160	1110 100 000 000 000 00 00 00 00 00 00	$\{ACD \to B, ABD \to C, BC \to A\}$
$[A] = 2, [B] = 3, [C] = 3, [D] = 3$		
no valid sets		
$[A] = 3, [B] = \infty, [C] = \infty, [D] = \infty$		
161	1000 000 000 000 000 00 00 00 00 00 00	$\{BCD \to A\}$
$[A] = 3, [B] = 3, [C] = \infty, [D] = \infty$		
162	1100 000 000 000 000 00 00 00 00 00 00	$\{BCD \to A, ACD \to B\}$
$[A] = 3, [B] = 3, [C] = 3, [D] = \infty$		
163	1110 000 000 000 000 00 00 00 00 00 00	$\{BCD \to A, ACD \to B, ABD \to C\}$
$[A] = 3, [B] = 3, [C] = 3, [D] = 3$		
164	1111 000 000 000 000 00 00 00 00 00 00	$\{BCD \to A, ACD \to B, ABD \to C, ABC \to D\}$

Table B.1: Sets of functional dependencies for the quaternary case

B.2 The Quinary Case

For five attributes, the number of closed functional dependency sets is 14 480 (see also Section 8.2.3). Instead of presenting all the sets, a list is given of the number of cases for different classes of attribute dimensions on Table B.2. Contradictory classes are indicated by the symbol ϟ.

Class					Number
[A]	[B]	[C]	[D]	[E]	of cases
∞	∞	∞	∞	∞	1
1	∞	∞	∞	∞	9

		Class			Number
[A]	[B]	[C]	[D]	[E]	of cases
1	1	∞	∞	∞	69
1	1	1	∞	∞	189
1	1	1	1	∞	207
1	1	1	1	1	32
1	1	1	1	2	131
1	1	1	1	3	25
1	1	1	1	4	1
1	1	1	2	∞	322
1	1	1	2	2	82
1	1	1	2	3	6
1	1	1	2	4	↯ 0
1	1	1	3	∞	45
1	1	1	3	3	↯ 0
1	1	1	3	4	↯ 0
1	1	1	4	∞	1
1	1	1	4	4	↯ 0
1	1	2	∞	∞	431
1	1	2	2	∞	762
1	1	2	2	2	327
1	1	2	2	3	80
1	1	2	2	4	1
1	1	2	3	∞	103
1	1	2	3	3	4
1	1	2	3	4	↯ 0
1	1	2	4	∞	1
1	1	2	4	4	↯ 0
1	1	3	∞	∞	47
1	1	3	3	∞	5
1	1	3	3	3	2
1	1	3	3	4	↯ 0
1	1	3	4	∞	↯ 0
1	1	3	4	4	↯ 0
1	1	4	∞	∞	1
1	1	4	4	∞	↯ 0
1	1	4	4	4	↯ 0
1	2	∞	∞	∞	144
1	2	2	∞	∞	866
1	2	2	2	∞	1355
1	2	2	2	2	435
1	2	2	2	3	241
1	2	2	2	4	2
1	2	2	3	∞	533

Class					Number
[A]	[B]	[C]	[D]	[E]	of cases
1	2	2	3	3	71
1	2	2	3	4	1
1	2	2	4	∞	4
1	2	2	4	4	↯ 0
1	2	3	∞	∞	233
1	2	3	3	∞	105
1	2	3	3	3	12
1	2	3	3	4	↯ 0
1	2	3	4	∞	1
1	2	3	4	4	↯ 0
1	2	4	∞	∞	2
1	2	4	4	∞	↯ 0
1	2	4	4	4	↯ 0
1	3	∞	∞	∞	32
1	3	3	∞	∞	49
1	3	3	3	∞	46
1	3	3	3	3	18
1	3	3	3	4	1
1	3	3	4	∞	1
1	3	3	4	4	↯ 0
1	3	4	∞	∞	1
1	3	4	4	∞	↯ 0
1	3	4	4	4	↯ 0
1	4	∞	∞	∞	1
1	4	4	∞	∞	↯ 0
1	4	4	4	∞	↯ 0
1	4	4	4	4	↯ 0
2	∞	∞	∞	∞	14
2	2	∞	∞	∞	207
2	2	2	∞	∞	1070
2	2	2	2	∞	1411
2	2	2	2	2	451
2	2	2	2	3	419
2	2	2	2	4	7
2	2	2	3	∞	1171
2	2	2	3	3	287
2	2	2	3	4	8
2	2	2	4	∞	19
2	2	2	4	4	1
2	2	3	∞	∞	646
2	2	3	3	∞	557
2	2	3	3	3	112

Class					Number of cases
[A]	[B]	[C]	[D]	[E]	
2	2	3	3	4	4
2	2	3	4	∞	15
2	2	3	4	4	↯ 0
2	2	4	∞	∞	14
2	2	4	4	∞	1
2	2	4	4	4	↯ 0
2	3	∞	∞	∞	91
2	3	3	∞	∞	278
2	3	3	3	∞	288
2	3	3	3	3	96
2	3	3	3	4	8
2	3	3	4	∞	18
2	3	3	4	4	1
2	3	4	∞	∞	13
2	3	4	4	∞	1
2	3	4	4	4	↯ 0
2	4	∞	∞	∞	4
2	4	4	∞	∞	1
2	4	4	4	∞	↯ 0
2	4	4	4	4	↯ 0
3	∞	∞	∞	∞	4
3	3	∞	∞	∞	22
3	3	3	∞	∞	60
3	3	3	3	∞	70
3	3	3	3	3	23
3	3	3	3	4	7
3	3	3	4	∞	14
3	3	3	4	4	2
3	3	4	∞	∞	11
3	3	4	4	∞	4
3	3	4	4	4	1
3	4	∞	∞	∞	3
3	4	4	∞	∞	2
3	4	4	4	∞	1
3	4	4	4	4	↯ 0
4	∞	∞	∞	∞	1
4	4	∞	∞	∞	1
4	4	4	∞	∞	1
4	4	4	4	∞	1
4	4	4	4	4	1

Table B.2: Nr. of quinary FD sets