Decomposition of Relational Schemata into Components Defined by Both Projection and Restriction

Extended Abstract

Stephen J. Hegner
Department of Computer Science and Electrical Engineering
Votey Building
University of Vermont
Burlington, VT 05405

(802)656-3330
hegner%uvm.edu@csnet-relay
..!{decvax,ihnp4}!dartvax!uvm-gen!hegner

This paper appeared in *Proceedings of the Seventh SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Austin, Texas, 21-23 March 1988, pp. 174-183..

(Corrected Version)

ABSTRACT

A generalized approach to the decomposition of relational schemata is developed in which the component views may be defined using both restriction and projection operators, thus admitting both horizontal and vertical decompositions. The realization of restrictions is enabled through the use of a Boolean algebra of types, while true independence of projections is modelled by permitting null values in the base schema. The flavor of the approach is algebraic, with the the collection of all candidate views of a decomposition modelled within a lattice-like framework, and the actual decompositions arising as Boolean subalgebras. Central to the framework is the notion of *bidimensional join dependency*, which generalizes the classical notion of join dependency by allowing the components of the join to be selected horizontally as well as vertically. Several properties of such dependencies are presented, including a generalization of many of the classical results known to be equivalent to schema acyclicity. Finally, a characterization of the nature of dependencies which participate in decompositions is presented. It is shown that there are two major types, the bidimensional join dependencies, which are tuple generating and allow tuple removal by implicit encoding of knowledge, and splitting dependencies, which simply partition the database into two components.

0. Introduction

With the development of general properties of join dependencies to characterize reconstructibility of a schema from the components of its decomposition [AhBU79, BeVa81, Scio82], separability to characterize independence of the components [ChMe87], and acyclicity [BFMY83] to characterize complexity, the topic of decomposition of relational schema is approaching maturity. Nonetheless there remain important extensions which have yet to be considered in detail. It is the purpose of the research described here to provide extensions of relational decomposition theory to such situations. There are three major parts to this work, as outlined below.

Database schemata are mathematical objects. Therefore, it seems reasonable to us that the theory of database schema decomposition should be embeddable in a mathematical framework which supports decomposition. In Section 1, we show how to embed the problem of schema decomposition into the context of decomposition of bounded weak partial lattices, by representing a view by its kernel. The roots of this work are in that of Bancilhon and Spyratos [BaSp81a, BaSp81b], as well as our own earlier work [Hegn83, Hegn84].

The principles developed in Section 1 are very general, and although they provide much insight on what a decomposition theory must look like, they cannot by themselves provide one. The class of all views of a schema is just too large a set upon which to found a productive theory. In Section 2, we develop a theory of decompositions based upon the restriction operator. The framework differs from the traditional one in that we employ a Boolean algebra of types, rather than a flat set of disjoint ones. In this regard we have followed the pioneering work of McSkimin and Minker [McMi79] and Reiter [Reit80, Reit84]. With the current rapprochement of database management with knowledge representation in general [Kers86, BrMy86] and with logic programming in general [Mink87], such a framework seems to be increasingly relevant. Indeed, researchers at MCC have recently proposed an extension to Prolog in which such a hierarchy is central [AïNa86].

Although vertical decomposition in terms of projections is the dominant mode, there are strong arguments in favor of establishing a systematic theory of *horizontal decompositions* as well, defined in terms of *restrictions*. Indeed, in addition to the desirability of such a theory in its own right, as argued by Smith a decade ago [Smit78], such decompositions are central to the data distribution policies of many distributed database management systems. The *Gamma* dataflow DBMS [DGKG86] is but one recent example which explicitly identifies horizontal partitioning of the database as central to its storage organization.

In our earlier work [Hegn83, Hegn84], we attempted to model all decompositions as restrictions, arguing that the horizontal mode of decomposition completely subsumed the vertical. While this approach was technically solid, it did not seem to be conceptually satisfactory. This shortcoming has been rectified here. In Section 2, we outline a unified approach to restrictive decomposition which encompasses both the horizontal and vertical modes. The cornerstone of the approach is to explicitly incorporate null values into the framework by extending the hierarchy of types to have particular varieties of incomplete information nulls. Projection then amounts to a restriction in which certain attributes must carry null values.

In order to convincingly demonstrate that the framework developed in Section 2 is useful, it is

important that we demonstrate that it not only recaptures but also extends the traditional results. In Section 3, we provide two major classes of results in that direction. First, we formalize the notion of bidimensional join dependency, which generalizes the classical notion by allowing horizontal decompositions across types as well as the traditional vertical or projective decompositions. It is also shown how such a dependency supports the decomposition of a schema in a canonical way. The second contribution in Section 3 regarding join dependencies generalizes the notion of acyclicity. We demonstrate that several of the more common characterizations of acyclicity extend to this more general class of dependency.

The format of this paper is one of an extended abstract. Although we have tried to be technically precise, for the most part we have omitted proofs and and more detailed discussions. A much more thorough development of many of the results will appear in the forthcoming monograph [Hegn88]; we anticipate that others will appear in expanded form in the near future as well.

1. Algebraic Structure of Views and Decomposition

Since our approach to decomposition is an algebraic one, it is important to begin with a sketch of the basic algebraic principles which underly this approach. Throughout, we assume a basic familiarity with those aspects of algebra related to partial orders, lattices, Boolean algebras, and the like. The references [Grät78] and [Birk67] provide detailed elaborations of the appropriate topics. We also assume familiarity with the basic ideas and notation of relational database theory, such as presented in [Maie83].

1.1 Schemata, Views, and Decompositions

1.1.1 Relational schemata and mappings A *relational schema* is a pair $\mathbf{D} = (\mathsf{Rel}(\mathbf{D}), \mathsf{Con}(\mathbf{D}))$ in which $\mathsf{Rel}(\mathbf{D})$ is a set of *relation names* and $\mathsf{Con}(\mathbf{D})$ is a set of constraints. Each relation name R has associated with it an *arity* $\mathsf{Arity}(R)$. Relative to a domain $\mathcal{D}$, a *database* over $\mathbf{D}$ is a function I which assigns to each $R \in \mathsf{Rel}(\mathbf{D})$ a relation R^I of arity $\mathsf{Arity}(R)$. The set of all databases of $\mathbf{D}$ relative to a fixed domain $\mathcal{D}$ is denoted $\mathsf{DB}(\mathbf{D})$. For now, we make no particular assumptions regarding the nature of $\mathsf{Rel}(\mathbf{D})$ or $\mathsf{Con}(\mathbf{D})$, save that the latter identify a subset of $\mathsf{DB}(\mathbf{D})$, termed the *legal databases* of $\mathbf{D}$ and denoted $\mathsf{LDB}(\mathbf{D})$.

Database mappings between schemata may be defined in a number of ways, such as logical interpretations using the *relational calculus*, or as combinations of relational operators using the *relational algebra*. Regardless of the method of definition, however, a database mapping $f : \mathbf{D} \to \mathbf{E}$ defines an underlying function $f^* : \mathsf{DB}(\mathbf{D}) \to \mathsf{DB}(\mathbf{E})$ on the states of the schemata. f is *legal* if $f^*(\mathsf{LDB}(\mathbf{D})) \subseteq \mathsf{LDB}(\mathbf{E})$. If f is legal, we let $f' : \mathsf{LDB}(\mathbf{D}) \to \mathsf{LDB}(\mathbf{E})$ denote the appropriate restriction of f^*.

1.1.2 Views A *view* of the schema $\mathbf{D}$ is a pair $\Gamma = (\mathbf{V}, \gamma)$ in which $\mathbf{V}$ is a database schema (called the *view schema*, and $\gamma : \mathbf{D} \to \mathbf{V}$ is a legal database mapping for which $\gamma' : \mathsf{LDB}(\mathbf{D}) \to$

$\mathsf{LDB}(\mathbf{V})$ is surjective. Two special schemata are the *identity view* $\Gamma_\top(\mathbf{D})$ whose underlying mapping preserves the state of $\mathbf{D}$ identically, and the *zero view* $\Gamma_\bot(\mathbf{D})$ which collapses all states of $\mathbf{D}$ to the same view state.

1.1.3 The decomposition mapping Given a set $\mathcal{V}$ of views of $\mathbf{D}$ and $X = \{\Gamma_1 = (\mathbf{V}_1, \gamma_1), \Gamma_2 = (\mathbf{V}_2, \gamma_2), .., \Gamma_n = (\mathbf{V}_n, \gamma_n)\} \subseteq \mathcal{V}$, define the *decomposition function* $\Delta(X) : \mathsf{LDB}(\mathbf{D}) \to \mathsf{LDB}(\mathbf{V}_1) \times \mathsf{LDB}(\mathbf{V}_2) \times .. \times \mathsf{LDB}(\mathbf{V}_n)$ by $s \mapsto (\gamma'_1(s), \gamma'_2(s), .., \gamma'_n(s))$. Reconstructibility is recaptured by requiring that $\Delta(X)$ be injective, while independence is recaptured by mandating that it be surjective. We define X to be a *decomposition* of $\mathbf{D}$ in $\mathcal{V}$ precisely in the case that $\Delta(X)$ is bijective.

1.2 Algebraic Formulation of Decomposition

1.2.1 Information content of a view Intuitively, the notion of information content of a represents the knowledge about the state of the base schema $\mathbf{D}$ which the view retains. The information content of a view is recaptured formally via the notion of the kernel of its defining function. The *kernel* of an arbitrary function $f : A \to B$ is the equivalence relation $\equiv_f = \{(x,y) \mid f(x) = f(y)\}$. We extend this to views by defining the kernel of the view $\Gamma = (\mathbf{V}, \gamma)$ to be the equivalence relation $\equiv_{\gamma'} = \{(x,y) \mid \gamma'(x) = \gamma'(y)\}$. We write $\Gamma_1 \preceq \Gamma_2$ just in case $\ker(\Gamma_2) \subseteq \ker(\Gamma_1)$. Two views are *semantically equivalent* if they have identical kernels. The notation $[\Gamma]$ is used to denote the equivalence class of views having the same kernel as Γ, and, if $\mathcal{V}$ is a set of views, $[\![\mathcal{V}]\!]$ is used to represent the associated set of equivalence classes. Modulo this equivalence, we may identify a view with its kernel, and furthermore, if we extend the relation $\preceq$ to equivalence classes, it defines a partial order (which we also denote by $\preceq$) on the classes of semantically equivalent views. Observe also that the kernel of $\Gamma_\top(\mathbf{D})$ is the identity relation on $\mathsf{LDB}(\mathbf{D})$, which is the finest partition on $\mathsf{LDB}(\mathbf{D})$, and that the kernel of $\Gamma_\bot(\mathbf{D})$ is the trivial partition $\{\mathsf{LDB}(\mathbf{D})\}$, which is the coarsest.

1.2.2 View join The join of two equivalence classes of views identifies a third equivalence class whose information content is essentially the union of the information contents of the components. More precisely, the view join operator is realized as the supremum of the two associated partitions. That is, if Γ_1 and Γ_2 are views of $\mathbf{D}$, then $[\Gamma_3] = [\Gamma_1] \vee [\Gamma_2]$ precisely in the case that $\ker(\Gamma_3) = \sup\{\ker(\Gamma_1), \ker(\Gamma_2)\}$. The importance of this operation is identified in the following.

1.2.3 Proposition *Let $X = \{\Gamma_1, \Gamma_2, .., \Gamma_n\}$ be a set of views of $\mathbf{D}$. The decomposition map $\Delta(X)$ is injective if and only if $[\Gamma_1] \vee [\Gamma_2] \vee .. \vee [\Gamma_n] = [\Gamma_\top(\mathbf{D})]$.* □

1.2.4 View meet The meet of two equivalence classes of views identifies a third equivalence class whose information content is effectively the intersection of the information contents of the components. It is tempting to parrot the definition of view join, simply replacing "sup" with "inf". However, this does not quite recover the notion which we seek, as the following example illustrates.

1.2.5 Example Let **D** be the schema which has just two unary relation symbols, R and S, one domain, and the single constraint which states that no domain element can be in both relations. The sentence $(\forall x)(\neg R(x) \vee \neg S(x))$ formalizes this constraint. Let Γ_R be the view which retains just the R relation and discards S, and let Γ_S be defined analogously for S. Let $(r_1, s_1) \in \mathsf{LDB}(\mathbf{V}_R)$, and let $(r_2, s_2) \in \mathsf{LDB}(\mathbf{V}_S)$. Then $(r_1, s_1) \equiv_{\gamma_R} (r_1, \emptyset) \equiv_{\gamma_S} (\emptyset, \emptyset) \equiv_{\gamma_R} (\emptyset, s_2) \equiv_{\gamma_S} (r_2, s_2)$, so that $\inf\{\ker(\Gamma_R), \ker(\Gamma_S)\} = \{\mathsf{LDB}(\mathbf{D})\}$, Yet, it is clear that these two views are not independent.

There is apparently no reasonable way to define the concept of "common information" for two arbitrary views. As illustrated by the above example, the problem which arises is that the kernels of the views may not commute, so that the kernel operators may be composed repeatedly to collapse states which should be distinguished. To rectify this, it is necessary to directly impose the additional condition that the kernels commute; that is, that $\ker(\Gamma_1) \circ \ker(\Gamma_2) = \ker(\Gamma_2) \circ \ker(\Gamma_1)$, with "$\circ$" denoting ordinary relational composition. In this case, $\inf\{\ker(\Gamma_1), \ker(\Gamma_2)\} = \ker(\Gamma_1) \circ \ker(\Gamma_2)$. Thus, we are forced regard the meet as a partial operation, defining $[\Gamma_1] \wedge [\Gamma_2] = [\Gamma_3]$ only in the case that both $\ker(\Gamma_3) = \inf\{\ker(\Gamma_1), \ker(\Gamma_2)\}$, and $\ker(\Gamma_1) \circ \ker(\Gamma_2) = \ker(\Gamma_2) \circ \ker(\Gamma_1)$. If $\ker(\Gamma_1)$ and $\ker(\Gamma_2)$ do not commute, then the meet $[\Gamma_1] \wedge [\Gamma_2]$ is undefined.

It might seem that, armed with this definition, we may define the independence of $[\Gamma_1]$ and $[\Gamma_2]$ to be equivalent to $[\Gamma_1] \wedge [\Gamma_2] = [\Gamma_\perp(\mathbf{D})]$. While this definition is correct, it is not sufficient to define the independence of a set of three or more views using only pairwise independence, as the following example illustrates.

1.2.6 Example – the pairwise independence problem Let **D** have three unary relation symbols R, S, and T, the single constraint asserting that any domain element must be in either none of the relations, or else in exactly two of them. This condition is recaptured by the single sentence $(\forall x)(T(x) \Leftrightarrow ((R(x) \wedge \neg S(x)) \vee (\neg R(x) \wedge S(x))))$. Let Γ_R be the view which preserves the instance of R exactly, but discards that of S and T. Define Γ_S and Γ_T similarly. Then we clearly we have $[\Gamma_R] \wedge [\Gamma_S] = [\Gamma_R] \wedge [\Gamma_T] = [\Gamma_S] \wedge [\Gamma_T] = [\Gamma_\perp(\mathbf{D})]$. Yet we cannot assert that the three views are independent of one another. Indeed, if we set $\mathcal{V} = \{\Gamma_R, \Gamma_S, \Gamma_T, \Gamma_\top(\mathbf{D}), \Gamma_\perp(\mathbf{D})\}$ and ask whether $X = \{[\Gamma_R], [\Gamma_S], [\Gamma_T]\}$ is a decomposition, the answer must clearly be "no", as $\Delta(X)$ cannot be surjective. Indeed, the state of any one of the views is completely determined by that of the other two. In fact, any two element subset of X forms a decomposition which cannot be further refined. The proper check on a candidate set for decomposition must therefore be that, upon partitioning X into two classes and computing the join of each, the meet is still $[\Gamma_\perp(\mathbf{D})]$.

1.2.7 Proposition *Let $X = \{\Gamma_1, \Gamma_2, .., \Gamma_n\}$ be a set of views of* **D**. *The decomposition map $\Delta(X)$ is surjective if and only if for each partition $\{I, J\}$ of X, $\bigvee_{\Gamma \in I}[\Gamma] \wedge \bigvee_{\Gamma \in J}[\Gamma]$ exists and is equal to $[\Gamma_\perp(\mathbf{D})]$.* □

1.2.8 The appropriate algebraic structure Propositions 1.2.3 and 1.2.7 essentially state that each decomposition X of **D** is the set of atoms of a Boolean algebra, using the join and meet

operations just defined. However, our goal is not to identify such decompositions directly, but rather to extract them from a larger, albeit less well-behaved framework. On the one hand, the presence of join and meet operators strongly suggest that a lattice-like framework is an appropriate one. On the other hand, the observations made in 1.2.5 make it clear that to expect all meets to exist is to be unrealistically restrictive. Rather, the appropriate setting is a *bounded weak partial lattice*. Briefly, a bounded weak partial lattice is a quintuple $\mathbf{L} = (L, \vee, \wedge, \top, \bot)$ which looks exactly like a bounded lattice, except that the join "$\vee$" and meet operations "$\wedge$" are *partial* functions $L \times L \rightarrow L$, rather than total ones. We refer the reader to [Hegn88] or [Grät78, p. 41] for the technical details, which are simple conceptually but too lengthy to completely express here. The adjective *bounded* simply means that $\top$ is the greatest element and $\bot$ the least. The particular bounded weak partial lattice into which equivalence classes of views embed is $\mathbf{CPart}(S)$, whose underlying set $\mathsf{CPart}(S)$ is the set of all partitions on the set S. Join is just supremum, and meet is defined only for commuting partitions, in which case it is the composition. For more details on this structure, consult [Ore42].

1.2.9 Adequate sets of views Although it is not practical to expect all meets to be defined in a set $\mathcal{V}$ of views to be used as a basis for decomposition, it is quite reasonable to expect all joins to be represented. The reason is quite simple. If $X = \{\Gamma_1, \Gamma_2, .., \Gamma_n\}$ is a decomposition, we should be able to combine two of the views, say Γ_1 and Γ_2, into a single view Γ and regard $\{\Gamma, \Gamma_3, .., \Gamma_n\}$ to be another perfectly acceptable, albeit less refined, decomposition of $\mathbf{D}$. This motivates the following definition.

A set of views $\mathcal{V}$ of $\mathbf{D}$ is termed *adequate* if it satisfies the following three conditions.

(i) The identity view $\Gamma_\top(\mathbf{D}) \in \mathcal{V}$.
(ii) The zero view $\Gamma_\bot(\mathbf{D}) \in \mathcal{V}$.
(iii) For every $\Gamma_1, \Gamma_2 \in \mathcal{V}$, there is a $\Gamma_3 \in \mathcal{V}$ such that $[\Gamma_1] \vee [\Gamma_2] = [\Gamma_3]$.

1.2.10 Characterization theorem for decompositions *Let $\mathcal{V}$ be an adequate collection of views of* $\mathbf{D}$*, and let $[\![\mathcal{V}]\!]$ denote the set of semantic equivalence classes.*
(a) *$[\![\mathcal{V}]\!]$ has the structure of a bounded weak partial lattice, with $[\Gamma_\top(\mathbf{D})]$ the top element $\top$, $[\Gamma_\bot(\mathbf{D})]$ the bottom element $\bot$, and join $\vee$ and $\wedge$ as defined by view join and view meet above. (Note that $\vee$ will always be a total function, but $\wedge$ will only be partial, in general.) We denote this bounded weak partial lattice by* $\mathsf{Lat}([\![\mathcal{V}]\!])$.
(b) *The decompositions of* $\mathbf{D}$ *with components in $\mathcal{V}$ are in bijective correspondence with the full Boolean subalgebras of* $\mathsf{Lat}([\![\mathcal{V}]\!])$*. The atoms of such a Boolean subalgebra identify exactly (the semantic equivalence classes of) those views comprising the decomposition. (By a* full *Boolean subalgebra of* $\mathsf{Lat}([\![\mathcal{V}]\!])$*, we mean one which has the same top $\top$ and bottom $\bot$ as* $\mathsf{Lat}([\![\mathcal{V}]\!])$*). In this case, the components of the decomposition are exactly the atoms of the Boolean algebra.* □

1.2.11 The ordering of decompositions Decomposition X is *more refined* than decomposition Y if the semantic equivalence class of every view of Y is expressible as the join of some of the

semantic equivalence classes of views of X. We write $Y \leq X$ to denote this fact. Clearly, this condition is equivalent to stating that the Boolean algebra generated by Y is a subalgebra of that generated by X. A decomposition X is *maximal* if $X \leq Y$ implies that $[\![Y]\!] = [\![X]\!]$, and is *ultimate* if $Y \leq X$ for all decompositions Y.

1.2.12 Corollary *The schema* **D** *has an ultimate decomposition within* $\mathcal{V}$ *if and only if* $\mathsf{Lat}([\![\mathcal{V}]\!])$ *has a largest full Boolean subalgebra.* □

1.2.13 Example – A problem with very general view definitions One question which may be reasonably asked is whether any initial restriction should be placed upon the base set of views $\mathcal{V}$, or whether any view definable by first-order means should be allowed. The following example, which is closely related to 1.2.5, shows that the class of all views is too general. Let **D** have exactly two unary relation symbols R and S, with no constraints whatsoever. Let Γ_S be the view which preserves the state of R exactly, but discards that of S. Define Γ_S similarly. Now clearly, if we set $\mathcal{V} = \{\Gamma_R, \Gamma_S, \Gamma_\top(\mathbf{D}), \Gamma_\bot(\mathbf{D})\}$, then $X = \{\Gamma_R, \Gamma_S\}$ forms an ultimate decomposition of **D**. However, let us now define one additional view, Γ_T, with the single unary relation symbol T, defined by the formula $T(x) \Leftrightarrow ((R(x) \wedge \neg S(x)) \vee (\neg R(x) \wedge S(x))$. This view is added to $\mathcal{V}$. It is easy to see that any of the three sets $\{\Gamma_R, \Gamma_S\}$, $\{\Gamma_R, \Gamma_T\}$, and $\{\Gamma_S, \Gamma_T\}$ form maximal decompositions of **D**, yet cannot be further refined. Thus, by adding a "strange" view to an extremely simple schema, we lose the ability to factor into an ultimate decomposition. Therefore, in this work we examine a very special class of views, namely those defined by projection, for which this type of behavior may be controlled.

1.3 Discussion of Previous Work

Work on the algebraic aspects of partitions the partitions on a set predates any computer science; the comprehensive treatise is [Ore42]. The idea of utilizing the order-theoretic aspects of the kernels of views was first forwarded by Bancilhon and Spyratos [BaSp81a, BaSp81b]. In particular, they were the first to define view independence in terms of kernels which induce surjectivity of the decomposition mapping. However, they did not recognize the bigger picture into which such kernels fit, and so were unable to formulate decompositions themselves as algebraic objects (Boolean algebras) embedded within a larger object (the partial lattice of views).

It is interesting to note that in the more specific arena of decomposition using join dependencies, the notion of independence of schemata has had a long evolution. In the earliest formulations, such as [Riss77] and [Vard82], independence was characterized by simple join consistency or its appropriate generalization. In other words, it was taken as a starting point that inter-view constraints ensuring join consistency be allowed. Later, it was realized that the enforcement of such constraints effectively prohibits independent view update, and so the less strict notion of weak instance satisfaction was proposed [GrYa84]. In this case, the state of each view had to be consistent with the state of some base state, but the same base state need not work for all. However, this

approach, which avoided null values, posed some problems as well. Recently, Chan and Mendelzon [ChMe87] proposed a notion of independence which explicitly identifies the independence formulation of Bancilhon and Spyratos as the starting point. In adopting this perspective, they acknowledge that nulls are a necessary component of independent decompositions into projections with join reconstruction. In short, the specific case formulation has evolved to that which is suggested by the general framework. In the next section, we shall formulate an algebraic base for manipulating such nulls within this general framework.

2. The Structure of Restrict-Project Views

In this section, we develop the special properties of views defined by combinations of restriction and projection. A key feature of our development is that projection may be viewed as an extended form of restriction in an appropriately constructed framework. Therefore, the concepts are first developed for restrictive views only, and then extend to a combined restrict and project framework. For the sake of simplicity, we work exclusively within the framework of single relation schemata, although there is nothing essential about this assumption, and most of the results presented here may be expanded to a multirelational framework, at the expense of a more complicated notation.

2.1 Restrictive Views

In traditional database theory, the *restrict* operator, applied to a relation, selects only those tuples satisfying a certain condition. Upon accepting this definition, the immediate question becomes one of defining exactly how the restriction conditions are defined. In this work, the only restrictions allowed are those which restricts the members of a column to lie in a particular domain. To have such a definition be nontrivial, it is necessary to formalize the *interaction* of domains. This is accomplished by assigning to the domains the structure of a Boolean algebra. The definition of type algebra presented here differs from that which we utilized in earlier work [Hegn83, Hegn84] only in that all domains are taken to be finite. In this latter respect we follow the approach of Reiter [Reit84].

2.1.1 Definition A *type algebra* is a triple $\mathcal{T} = (\mathbf{T}, \mathbf{K}, \mathbf{A})$, where:
(a) **T** is a finite set of unary predicate symbols, called the *types*, which form a Boolean algebra under the logical operations of disjunction ($\vee$), conjunction ($\wedge$), and negation ($\neg$). The greatest element of this algebra is the universally true predicate $\top$, while the least element is the universally false predicate $\bot$. The *atomic types* are precisely the atoms of this Boolean algebra.
(b) **K** is a finite set of constant symbols, called the *names*.
(c) **A** is a set of sentences in the first-order language with equality whose relation symbols are those of **T**, and whose constant symbols are those of **K**. The axioms are strong enough to determine, for each $k \in \mathbf{K}$ and $\tau \in \mathbf{T}$, whether or not $\tau(k)$ is true. The axioms also define a *domain closure* condition for each $\tau \in \mathbf{T}$ which specifies that the constants of type τ are the only objects of that type.

For $a \in \mathbf{K}$, the *base type* of a is the least type $\tau \in \mathbf{T}$ such that $\mathbf{A} \models \tau(a)$, and is denoted $\mathsf{BaseType}(a)$. More generally, for any type υ, we say that say that a is of type υ if $\mathbf{A} \models \upsilon(a)$. Clearly, this is the case iff $\mathsf{BaseType}(a) \leq \upsilon$.

2.1.2 Database schemata and instances A *relational schema* $\mathbf{D} = (\mathsf{Rel}(\mathbf{D}), \mathsf{Con}(\mathbf{D}))$ over $\mathcal{T} = (\mathbf{T}, \mathbf{K}, \mathbf{A})$ has the property that $\mathsf{Con}(\mathbf{D})$ is a set of first-order sentences in the language including precisely the symbols of $\mathsf{Rel}(\mathbf{D})$ and $\mathcal{T}$, plus equality. We always assume that $\mathsf{Con}(\mathbf{D}) \models \mathbf{A}$. A *basis* for $\mathsf{Con}(\mathbf{D})$ is any set Σ with $\Sigma \cup \mathbf{A} \models \mathsf{Con}(\mathbf{D})$. The collection of all relational schemata over $\mathcal{T}$ is denoted $\mathsf{Sch}(\mathcal{T})$.

In this work, we always assume that $\mathsf{Rel}(\mathbf{D})$ contains exactly one relation symbol R, whose arity is denoted by n. We further assume that the columns of R are denoted by an attribute set $\mathbf{U} = \{A_1, A_2, .., A_n\}$, with A_i associated with the i^{th} column.

Databases are taken over the domain $\mathcal{D} = \mathbf{K}$. In effect, since we have postulated domain closure in $\mathcal{T}$, we may regard all tuples of relations to be comprised of constant names from $\mathbf{K}$. Note that since $\mathbf{K}$ is a finite set, all databases will be finite. This assumption has the advantage of completely putting aside any difficult model theoretic questions, such as whether or not views are first-order axiomatizable. On the other hand, it also puts aside many interesting and deep questions in the database and model theory. At the expense of some added machinery, most of the results presented here may be extended into a framework allowing infinite databases, but we do not pursue such issues here. For a development of some of the issues and problems with infinite domains, see [Vard82].

2.1.3 Modelling restrictions To formalize the notion of a database mapping defined by restriction, we begin with a definition at the level of tuples. A simple n-type over $\mathcal{T}$ is an n-tuple $t = (\tau_1, \tau_2, .., \tau_n)$ such that each $\tau_i \in \mathbf{T} \setminus \{\bot\}$. The collection of all simple n-types is denoted $\mathsf{Simple}(\mathcal{T}, n)$. The collection of all sets of n-tuples of elements of $\mathbf{K}$ is the powerset $\mathcal{P}(\mathbf{K}^n)$. The *restriction associated with t* is the function $\rho\langle t \rangle : \mathcal{P}(\mathbf{K}^n) \to \mathcal{P}(\mathbf{K}^n)$ given by $X \mapsto \{(x_1, x_2, .., x_n) \in X \mid x_i \text{ is of type } \tau_i, 1 \leq i \leq n\}$.

A *compound n-type over* $\mathcal{T}$ is a (possibly empty) set $S = \{s_1, s_2, .., s_k\}$ of simple n-types. The set of all compound n-types over $\mathcal{T}$ is denoted by $\mathsf{Compound}(\mathcal{T}, n)$. The *restriction* associated with S is denoted by $\rho\langle S \rangle$ and is given on elements by $X \mapsto \bigcup_{i=1}^{k} \rho\langle s_i \rangle(X)$. In other words, for a compound restriction, we simply take the union of the comprising simple restrictions. It is convenient to represent such restrictions by their underlying components using a summation notation; we will often express the compound n-tuple S as $\sum_{i=1}^{k} \rho\langle s_i \rangle$ when we wish to emphasize that it is representing a restriction, and will even write $\rho\langle S \rangle = \sum_{i=1}^{k} \rho\langle s_i \rangle$, although it must be remembered that the right-hand side is a generally nonunique *representation* of of the restriction, and more formally denotes a compound n-type. The set of all such representations is denoted $\mathsf{Restr}(\mathcal{T}, n)$. If $T = \{t_1, t_2, .., t_m\} \in \mathsf{Compound}(\mathcal{T}, n)$ also, then the *sum* $\rho\langle S \rangle + \rho\langle T \rangle$ is given by $\sum_{i=1}^{k} \rho\langle s_i \rangle + \sum_{j=1}^{m} \rho\langle t_j \rangle$. The *composition* $\rho\langle S \rangle \circ \rho\langle T \rangle$ is given by $\sum_{i=1}^{k} \sum_{j=1}^{m} \rho\langle s_i \rangle \circ \rho\langle t_j \rangle$.

2.1.4 Basis of a restriction It is convenient to be able to represent a restriction in terms of its primitive components. A simple n-type $t = (\tau_1, \tau_2, .., \tau_n)$ is *atomic* if each τ_i is an atomic type. $\mathsf{Atomic}(\mathcal{T}, n)$ denotes the set of all atomic simple types. The *basis* of a simple n-type $s = (\sigma_1, \sigma_2, .., \sigma_n)$ is $\{(\tau_1, \tau_2, .., \tau_n) \in \mathsf{Atomic}(\mathcal{T}, n) \mid \tau_i \leq \sigma_i\}$. The basis of a compound n-type S is the union of the bases of its constituents, and is denoted $\mathsf{Basis}(S)$. A compound n-type S is *primitive* if each $s \in S$ is atomic. We denote the class of all primitive n-types by $\mathsf{Primitive}(\mathcal{T}, n)$. As $\mathsf{Primitive}(\mathcal{T}, n)$ is just the set of all subsets of $\mathsf{Atomic}(\mathcal{T}, n)$, it clearly admits the structure of a Boolean algebra under union, intersection, and set complement. We call this algebra the *primitive restriction algebra*. The importance of this algebra relative to the behavior of projections is captured by the following proposition.

2.1.5 Proposition *The following conditions are equivalent.*

(i) $\mathsf{Basis}(T) \subseteq \mathsf{Basis}(S)$.

(ii) $(\forall x \in \mathcal{P}(\mathbf{K}^n))(\rho\langle T\rangle(x) \subseteq \rho\langle S\rangle(x))$.

(iii) $\ker(\rho\langle S\rangle) \subseteq \ker(\rho\langle T\rangle)$. □

Two compound n-types S and T are *basis equivalent* if they have the same basis. We write $\rho\langle S\rangle \equiv_* \rho\langle T\rangle$ to denote this, and write $[S]_*$ or $[\rho\langle S\rangle]_*$ to denote the associated equivalence class. The set of all compound types modulo this equivalence is denoted $[\mathsf{Restr}(\mathcal{T}, n)]_*$. When we wish to emphasize that these elements are over the database schema $\mathbf{D}$, we will write $[\mathsf{Restr}(\mathcal{T}, \mathbf{D})]_*$ instead. In view of the above proposition each each equivalence class is represented by a unique primitive n-type. Since we know that $\mathsf{Primitive}(\mathcal{T}, n)$ admits the structure of a Boolean algebra under the set-theoretic operations, this result essentially tells us that we may indifferently characterize the behavior of projections in two distinct ways, either as the size of the kernel, or else as the size of the image. We furthermore have the following representations.

2.1.6 Proposition *Let S and T be compound n-types. Then, within the Boolean algebra $[\mathsf{Restr}(\mathcal{T}, n)]_*$, the following hold.*

(a) $\rho\langle S\rangle \vee \rho\langle T\rangle = \rho\langle S\rangle + \rho\langle T\rangle$.

(b) $\rho\langle S\rangle \wedge \rho\langle T\rangle = \rho\langle S\rangle \circ \rho\langle T\rangle$. □

2.1.7 Equivalence classes A second equivalence relations arises when we consider the constraints on $\mathbf{D}$. In this case, we only work with legal states, so restrictions are regarded as mappings $\mathsf{LDB}(\mathbf{D}) \to \mathsf{DB}(\mathbf{D})$. This equivalence relation is denoted $\equiv_\dagger$, and is termed the *semantic equivalence* on $\mathsf{Restr}(\mathcal{T}, \mathbf{D})$. The set of all such equivalence classes is denoted $[\mathsf{Restr}(\mathcal{T}, \mathbf{D})]_\dagger$. This notation is also applied to individual members $\mathbf{S} \in \mathsf{Compound}(\mathcal{T}, \mathbf{D})$; $[\mathbf{S}]_*$ or $[\rho\langle\mathbf{S}\rangle]_*$ denotes the syntactic equivalence class of $\mathbf{S}$, and $[\mathbf{S}]_\dagger$ or $[\rho\langle\mathbf{S}\rangle]_\dagger$ denotes its semantic equivalence class. Since $\equiv_\dagger$ is defined using the same functions, but on a smaller domain than $\equiv_*$, it is clear that $\equiv_* \subseteq \equiv_\dagger$, *i.e.*, syntactic equivalence is a refinement of semantic equivalence.

2.1.8 The view associated with a restriction Let us define the *surjectification* of a function $f : A \to B$ to be the function $\vec{f} : A \to f(A)$. In other words, we make the function surjective by restricting its range. To regard a restriction $\rho : \mathsf{LDB}(\mathbf{D}) \to \mathsf{DB}(\mathbf{D})$ as the underlying mapping of a view, we need to surjectify it. Thus, the view $\Gamma_\rho = (\mathbf{V}_\rho, \gamma_\rho)$ associated with ρ must have $\gamma_\rho = \vec{\rho}$. The "only" problem that we have with this approach is that the view schema $\mathbf{V}_\rho$ must be axiomatized in such a way that $\mathsf{LDB}(\mathbf{V}_\rho) = \rho(\mathsf{LDB}(\mathbf{D}))$. Unfortunately, this is a very nasty problem in general, and examples exist for which $\rho(\mathsf{LDB}(\mathbf{D}))$ is not axiomatizable by first order means. We have avoided this problem by fixing the underlying domain $\mathcal{D}$ to be finite. Since $\rho(\mathsf{LDB}(\mathbf{D}))$ is a finite set, it is clearly axiomatizable. Actually, in any view participating in a decomposition, the axiomatization of states will be first order, even with infinite domains. [Vard82] contains a further discussion of this issue. *At any rate, with this understanding, we shall simply identify restrictions with their associated views, and forego any formal translation between the two.*

2.1.9 Proposition – Adequacy of $\mathsf{Restr}(\mathcal{T}, \mathbf{D})$ *The class* $\mathsf{Restr}(\mathcal{T}, \mathbf{D})$ *forms an adequate set of views of* $\mathbf{D}$. *Additionally, the characterization of joins of 2.1.6(a) extends to semantic views, in the precise sense that* $[\rho\langle\mathbf{S}\rangle]_\dagger \vee [\rho\langle\mathbf{T}\rangle]_\dagger = [\rho\langle\mathbf{S}+\mathbf{T}\rangle]_\dagger$, *for any* $\mathbf{S}, \mathbf{T} \in \mathsf{Restr}(\mathcal{T}, \mathbf{D})$.

PROOF OUTLINE: Since each equivalence class of $[\mathsf{Restr}(\mathcal{T}, \mathbf{D})]_\dagger$ is immediately identifiable with the kernel of any representative, and since the identity and zero functions are represented, it inherits the structure of a bounded weak partial lattice from $\mathbf{CPart}(\mathsf{LDB}(\mathbf{D}))$. Thus, all we need establish is the join characterization. However, it is easy to see that first restricting two restrictions to $\mathsf{LDB}(\mathbf{D})$ and then taking the supremum of the resulting kernels is identical in result to first taking the supremum as restrictions, from which the characterization is immediate. □

2.2 Extension to Include Projection

Our strategy in modelling the projection operator is to employ the framework developed in 2.1 by regarding projection as a special form of restriction. To utilize projections while retaining the notion of independence of view independence in decompositions requires that we allow the presence of incomplete information nulls to "fill in the differences". We start by modelling null values as distinguished constant symbols in the underlying type algebra $\mathcal{T}$.

2.2.1 Definition Let $\mathcal{T} = (\mathbf{T}, \mathbf{K}, \mathbf{A})$ be a type algebra. The *associated null augmented algebra* of $\mathcal{T}$, denoted $\mathsf{Aug}(\mathcal{T}) = (\mathsf{Aug}(\mathbf{T}), \mathsf{Aug}(\mathbf{K}), \mathsf{Aug}(\mathbf{A}))$, is defined as follows.
(a) To $\mathbf{K}$ we add one new constant symbol ν_τ for each $\tau \in \mathbf{T} \setminus \{\bot\}$. ν_τ is called the *null of type* τ. Formally, $\mathsf{Aug}(\mathbf{K}) = \mathbf{K} \cup \mathbf{K}_\nu$, where $\mathbf{K}_\nu = \{\nu_\tau \mid \tau \in \mathbf{T} \setminus \{\bot\}\}$.
(b) To $\mathbf{T}$ we add one new atomic type ι_τ for each $\tau \in \mathbf{T} \setminus \{\bot\}$. The only symbol of type ι_τ is the null ν_τ, so that these new types are disjoint from all existing types. Of course, new nonatomic types will be constructed also.
(c) To $\mathbf{A}$ are added the axioms to make the conditions of (a) and (b) hold, including the necessary domain closure axioms and definitions of new nonatomic types.

There is one class of nonatomic types of $\mathrm{Aug}(\mathcal{T})$ which are particularly important. For any type τ of $\mathcal{T}$, we let $\widehat{\tau} = \tau \vee (\bigvee_{\tau \leq \upsilon} \iota_\upsilon)$. $\widehat{\tau}$ is called the *null completion* of τ.

Also, since we shall now use the symbol $\top$ to denote the universal type of $\mathrm{Aug}(\mathcal{T})$, the symbol $\top_{\bar{\nu}}$ will be used to denote the universal element of $\mathcal{T}$.

2.2.2 Subsumption of augmented tuples – the semantics of nulls Nulls are not *just* constant symbols, free to be interpreted in any way. Rather, they possess a very special semantics, which we now describe. Let $a = (a_1, a_2, .., a_n)$ and $b = (b_1, b_2, .., b_n)$ be instance n-tuples over the type algebra $\mathrm{Aug}(\mathcal{T})$. We say that *a subsumes b* (written $b \leq a$) if, for each i, $1 \leq i \leq n$, exactly one of the following conditions holds.

(i) $a_i = b_i$;

(ii) For some $\tau_1, \tau_2 \in \mathbf{T}$, $b_i = \nu_{\tau_2}$ and a_i is of type τ_1 and $\tau_1 \leq \tau_2$.

(iii) For some $\tau_1, \tau_2 \in \mathbf{T}$, $a_i = \nu_{\tau_1}$, $b_i = \nu_{\tau_2}$, and $\tau_1 \leq \tau_2$.

This concept of ordering tuples under null subsumption is similar to that given by Biskup [Bisk81], extended to include the type hierarchy.

A tuple a is *complete* if it is subsumed by no other tuple than itself. Two sets X and Y of n-tuples are *null-equivalent* if each $x \in X$ is subsumed by some $y \in Y$, and conversely. X is *null-complete* if for any Y which is null-equivalent to X, we have $Y \subseteq X$. Similarly, X is *null-minimal* if for any Y which is null-equivalent to X, we have $X \subseteq Y$. For any set X of instance n-tuples there is a unique null-complete set, denoted by $\hat{X}$ and termed the *null completion* of X (obtained by adding to X all n-tuples subsumed by existing members of X), and a unique null-minimal set, denoted $\check{X}$ (which is obtained by deleting from X all n-tuples which are subsumed by other tuples). X is *information complete* if $\check{X}$ consists entirely of complete tuples.

One model for a null type ν_τ occurring in an instance tuple is that the value for that position is not specified in that tuple. For example, in the tuple (a, b, ν_τ), the value of the third entry is not specified, but is only restricted to be of type τ. Thus, such a tuple may be viewed as representing the formula $\vee\{R(a,b,x) \mid \mathbf{A} \models \tau(x)\}$. If a tuple such as (a, b, c) is present in the relation also, with $\mathbf{A} \models \tau(c)$, then (a, b, ν_τ) is subsumed by it, and so carries no further information. However, this is only an interpretation of the formal model; there is no explicit use of incomplete information or multiple-literal clauses within this approach.

2.2.3 Concepts behind representation of projection Before presenting the formalities, we briefly motivate the underpinnings of our representation of projection with a simple example. Let $\mathbf{D}$ have just a single ternary relation symbol R; with the usual attribute notation we write $R[ABC]$. Assume further that the underlying algebra $\mathcal{T}$ has just one atomic type τ, and all three columns take their non-null values from that type. Consider the problem of representing the AB projection as a restriction. Rather than drop the C column entirely, as one would normally do, we retain this column but replace all values with nulls. Thus, we would like the mapping to look something like $(a, b, c) \mapsto (a, b, \nu_\top)$. Now if the database is null complete, then a simple restriction represented by a 3-type of the form $\pi\langle AB\rangle = (\top_{\bar{\nu}}, \top_{\bar{\nu}}, \iota_\top)$ will represent this projection. This is precisely why

we have formulated the notion of null-completeness. We must always be sure to enforce null completeness when requiring a proper semantic interpretation. *Therefore, unless explicitly stated to the contrary, the legal states of* **D** *will always be assumed to be null-complete.* Of course, this is only a modelling convention; an actual implementation would likely work with null-minimal states and compute the necessary nulls, as needed, from the subsumption conditions.

2.2.4 Concepts behind restrict-project mappings Conceptually, in a restrict-project mapping, we first apply a restriction operator, and then a projection. Let us continue with the example of the previous paragraph, but assume that $\mathcal{T}$ now admits at least three distinct types $\{\tau_1, \tau_2, \tau_3\}$. We want to first restrict A to τ_1, B to τ_2, and C to τ_3; this is succinctly represented by the restriction ρ_t, with $t = (\tau_1, \tau_2, \tau_3)$. Next, we want to project as per the previous example with $\pi\langle AB\rangle$. It is easy to see that we can model this situation by replacing $\pi\langle AB\rangle$ with $(\tau_1, \tau_2, \iota_{\tau_3})$.

2.2.5 Formalization of restrict-project mappings Because $\mathsf{Aug}(\mathcal{T})$ is a type algebra, all of the results of 2.1 apply with $\mathcal{T}$ replaced by $\mathsf{Aug}(\mathcal{T})$. However, this framework is too big in that it contains semantically meaningless restrict-project mappings. Thus, we must identify the appropriate constraints, motivated by the above examples. In the following, we fix a type algebra $\mathcal{T}$ and a database schema **D** over $\mathsf{Aug}(\mathcal{T})$.

The *projective types* of $\mathsf{Aug}(\mathcal{T})$ are those members of the set $\Pi(\mathcal{T}) = \{\iota_\tau \mid \tau \in \mathbf{T}\} \cup \{\top_{\bar{\nu}}\}$. The *restrictive types* of $\mathsf{Aug}(\mathcal{T})$ are defined by the set $\mathrm{P}(\mathcal{T}) = \{\widehat{\tau} \mid \tau \in \mathbf{T}\}$. A *restrict-project type* (or $\pi{\cdot}\rho$-*type* is either a restrictive type or a projective type.

A *simple* ρ *n-type* (resp. *simple* π *n-type) is a simple n-type* $u = (\upsilon_1, \upsilon_2, .., \upsilon_n)$ *over* $\mathsf{Aug}(\mathcal{T})$ *with the property that each* υ_i *is a restrictive type (resp. projective type). A* simple $\pi{\cdot}\rho$ *mapping* is a composition f of the form $\zeta \circ \upsilon$ in which ζ is a simple projective type and υ is a simple restrictive type. For $X \subseteq \mathbf{U}$ and $t = (\tau_1, \tau_2, .., \tau_n) \in \mathsf{Simple}(\mathcal{T}, n)$, we will often use the notation $\pi\langle X\rangle \circ \rho\langle t\rangle$ to denote the simple $\pi{\cdot}\rho$ mapping whose restrictive component is $(\widehat{\tau_1}, \widehat{\tau_2}, .. \widehat{\tau_n})$. The projective component of this mapping is $(y_1, y_2, .., y_n)$, with $y_i = \top_{\bar{\nu}}$ if $A_i \in X$ and $y_i = \iota_{\tau_i}$ otherwise. A *compound* $\pi{\cdot}\rho$ *n-type* is a set of simple $\pi{\cdot}\rho$-types. Given a compound $\pi{\cdot}\rho$ *n*-type S, the *restrict-project mapping* (or $\pi{\cdot}\rho$-*mapping*) associated with S is just the restriction associated with S (in the sense of 2.1.3), and is denoted $\rho\langle S\rangle$ also. Representations are defined as in 2.1.3 as well, with the set of all representations of restrict-project-filter mappings denoted $\mathsf{RestrProj}(\mathcal{T}, n)$. It is important to observe that $\mathsf{RestrProj}(\mathcal{T}, n) \subseteq \mathsf{Restr}(\mathsf{Aug}(\mathcal{T}), n)$, but that inclusion is in general proper. For a database schema **D** over $\mathsf{Aug}(\mathcal{T})$, $\mathsf{RestrProj}(\mathcal{T}, \mathbf{D})$ is defined analogously.

2.2.6 Extended schemata and their views An *extended database schema* **D** over $\mathcal{T}$ is a database schema over $\mathsf{Aug}(\mathcal{T})$ with the property that each member of $\mathsf{LDB}(\mathbf{D})$ is null-complete. On such a schema, the equivalence classes on $\mathsf{RestrProj}(\mathcal{T}, \mathbf{D})$ induced by the relation $\equiv_\dagger$ (as defined on $\mathsf{Restr}(\mathsf{Aug}(\mathcal{T}), \mathbf{D})$), are denoted $[\mathsf{RestrProj}(\mathcal{T}, \mathbf{D})]_\dagger$. As explained in 2.1.8, we shall regard such project-restrict mappings as views on **D**.

2.2.7 Proposition – Adequacy of $\mathsf{RestrProj}(\mathcal{T},\mathbf{D})$ *Let* $\mathbf{D}$ *be an extended null-complete database schema over* $\mathcal{T}$. *Then the class* $\mathsf{RestrProj}(\mathcal{T},\mathbf{D})$ *forms an adequate set of views of* $\mathbf{D}$. *The characterization of joins of 2.1.6(a) (with* $\mathcal{T}$ *replaced by* $\mathsf{Aug}(\mathcal{T})$*) applies as it does in 2.1.10, in the precise sense that* $[\rho\langle\mathbf{S}\rangle]_\dagger \vee [\rho\langle\mathbf{T}\rangle]_\dagger = [\rho\langle\mathbf{S}+\mathbf{T}\rangle]_\dagger$, *for any* $\mathbf{S},\mathbf{T} \in \mathsf{RestrProj}(\mathcal{T},\mathbf{D})$.

PROOF OUTLINE: The proof proceeds almost as in 2.1.10. The only additional item necessary is to verify that the sum of two $\pi\cdot\rho$-mappings is a $\pi\cdot\rho$-mapping, but this follows immediately from the definition. □

3. Characterization of Decomposition

3.1 Generalized Projective Decomposition

In the traditional theory, projective decompositions are governed by join dependencies. In this section, we generalize the notion of join dependency to recapture both vertical and horizontal decomposition.

3.1.1 Definition – bidimensional join dependencies Let k be a positive integer, let $X = \bigcup\{X_1, X_2, .., X_k\}$ with each $X_i \subseteq \mathcal{P}(\mathbf{U})$, and let $T = \{t_1, t_2, .., t_k, t\} \subseteq \mathsf{Simple}(\mathcal{T}, n)$, with $t_i = (\tau_{i1}, \tau_{i2}, ..\tau_{ik})$ and $t = (\tau_1, \tau_2, .., \tau_n)$. We wish to characterize the constraints necessary to ensure a *projective decomposition* governed by the following rule.

$$\sum_{i=1}^{k} \pi\langle X_i\rangle \circ \rho\langle t_i\rangle \rightsquigarrow \pi\langle X\rangle \circ \rho\langle t\rangle$$

The symbol "$\rightsquigarrow$" denotes that the view defined by the left-hand side is sufficient to determine that of the right-hand side, so that the latter need not be explicitly stored. Rather, it may be computed as needed. If $X = \mathbf{U}$ and $t = (\top_{\bar{\mathrm{v}}}, \top_{\bar{\mathrm{v}}}, .., \top_{\bar{\mathrm{v}}})$, then the above reduces to a decomposition of the entire database. If each $t_i = (\top_{\bar{\mathrm{v}}}, \top_{\bar{\mathrm{v}}}, .., \top_{\bar{\mathrm{v}}})$ as well, then this reduces to specifying an ordinary join decomposition $\sum_{i=1}^{k} \pi\langle X_i\rangle \rightsquigarrow \mathbf{1}$, or, in the more common notation, $\bowtie[X_1, X_2, .., X_k] = \mathbf{1}$.

We wish to define the *bidimensional join dependency* $\bowtie[X_1\langle t_1\rangle, X_2\langle t_2\rangle, .., X_k\langle t_k\rangle]\langle t\rangle$ which will support such a decomposition. Let $\Lambda(X_i, t_i)$ denote the formula $R(z_1, z_2, .., z_n)$ in which $z_j = x_j$ if $A_j \in X_i$ and $z_j = \nu_{\tau_{ij}}$ otherwise, and define $\Lambda(X, t)$ similarly. Let β_i be $\tau_i(x_i)$ if $A_i \in X$ and $\iota_{\tau_i}(x_i)$ otherwise. Define the bidimensional join dependency $J = \bowtie[X_1\langle t_1\rangle, X_2\langle t_2\rangle, .., X_k\langle t_k\rangle]\langle t\rangle$ by the following formula.

$$(\forall x_1, x_2, .., x_n)((\beta_1 \wedge \beta_2 \wedge .. \wedge \beta_n \wedge \Lambda(X_1, t_1) \wedge \Lambda(X_2, t_2) \wedge .. \wedge \Lambda(X_k)) \Leftrightarrow \Lambda(X, t)) \qquad (*)$$

Extending the terminology of Sciore [Scio80], define the *objects* of J to be $\mathsf{Objects}(J) = \{X_i\langle t_i\rangle \mid 1 \leq i \leq k\}$. The i^{th} *component view* of J is that defined by $\pi\langle X_i\rangle \circ \rho\langle t_i\rangle$, while the *target view* is that identified by $\pi\langle X\rangle \circ \rho\langle t\rangle$. In analogy to the traditional case, we call J *vertically full* if $\mathsf{Span}(X) = \mathbf{U}$, and *horizontally full* if $t = (\top_{\bar{\mathrm{v}}}, \top_{\bar{\mathrm{v}}}, .., \top_{\bar{\mathrm{v}}})$. If $k = 2$, J is termed a *bidimensional multivalued dependency*. In case any of the t_i's or t are equal to $(\top_{\bar{\mathrm{v}}}, \top_{\bar{\mathrm{v}}}, .., \top_{\bar{\mathrm{v}}})$, we may omit them in the notation.

In their most general form, bidimensional join dependencies may be very complex. To illustrate their use, we will examine a few particularly important examples.

3.1.2 Vertical join dependencies We may recapture the traditional case of vertical dependencies by setting $t = t_1 = t_2 = .. = t_k$. It is easy to see by the null subsumption rules that in this case the "$\Leftrightarrow$" in the above formula may be replaced with "$\Rightarrow$", as is the more standard representation in terms of implicational dependencies [Fagi82]. If we further take $t = t_1 = t_2 = .. = t_k = (\top_{\bar{\nu}}, \top_{\bar{\nu}}, .., \top_{\bar{\nu}})$, then we recapture the case of no horizontal embedding at all. The following example expands upon this case.

3.1.3 Example Let us develop a few details of the example $\bowtie[AB, BC, CD, DE]$ of the schema $R[ABCDE]$ within our framework. First, here is the defining formula.

$$
\begin{aligned}
&(\forall x_1, x_2, x_3, x_4, x_5) \\
&\qquad ((\top_{\bar{\nu}}(x_1) \wedge \top_{\bar{\nu}}(x_2) \wedge \top_{\bar{\nu}}(x_3) \wedge \top_{\bar{\nu}}(x_4) \wedge \top_{\bar{\nu}}(x_5) \wedge \\
&\qquad R(x_1, x_2, \nu_\top, \nu_\top, \nu_\top) \wedge R(\nu_\top, x_2, x_3, \nu_\top, \nu_\top) \wedge \\
&\qquad R(\nu_\top, \nu_\top, x_3, x_4, \nu_\top) \wedge R(\nu_\top, \nu_\top, \nu_\top, x_4, x_5)) \\
&\qquad \Rightarrow R(x_1, x_2, x_3, x_4, x_5)).
\end{aligned}
$$

This formula is not different than the classical join constraint, except that we use nulls in those positions which occur on the left-hand side but not the right. Because of null-completion, we are assured of the existence of such nulls. However, replacing those nulls with arbitrary universally quantified variables would not change the semantics in this case, so this is exactly a classical join.

It is important to note that some of the inference rules for join dependencies which hold in the traditional setting ([BeVa81]) do not hold in this null-augmented one. For example, it is not the case that $\bowtie[AB, BC, CD, DE] \models \bowtie[AB, BC]$. This will be true even if we replace each $\nu_\top$ with a variable of type $\top$ in the formula above. Similarly, $\bowtie[BC, CD]$ and $\bowtie[CD, DE]$ are not a consequence of $\bowtie[AB, BC, CD, DE]$. This cannot be corrected without allowing nulls in the joined columns. The only recourse is to explicitly postulate the enforcement of these dependencies. However, it is the case, under the assumption of null completeness, that $\{\bowtie[AB, BC], \bowtie[BC, CD], \bowtie[CD, DE]\} \models \bowtie[AB, BC, CD, DE]$.

The dependencies $\bowtie[AB, BCDE]$, $\bowtie[ABC, CDE]$, and $\bowtie[ABCD, DE]$ are consequences of $\bowtie[AB, BC, CD, DE]$. However, there are other problems associated with them, which will be identified below.

3.1.4 Horizontal join dependencies In our earlier work [Hegn83, Hegn84], we advocated modelling projective decompositions by utilizing built in "placeholder" nulls. This type of decomposition is also recaptured here. Consider the schema $R[ABC]$ in which there are two atomic types. τ_1 identifies the "normal" domains of all attributes, while τ_2 is the placeholder null type; there is exactly one constant η_2 of the latter type, which we term the *placeholder null*. We wish to recapture the traditional dependency $AB \bowtie BC$ in a placeholder fashion. Values of type τ_2 are

allowed for attributes *A* and *C*, but not for *B*. The governing bidimensional join dependency is $\bowtie[AB\langle\tau_1,\tau_1,\tau_2\rangle, BC\langle\tau_2,\tau_1,\tau_1\rangle]\langle\tau_1,\tau_1,\tau_1\rangle$. Thus, a tuple of the form (a,b,c) is in the database if and only if the tuples (a,b,η_2) and (η_2,b,c) are. Since η_2 is the only constant of type τ_2, it may be unambiguously replaced with ν_{τ_2}. Notice here that the nulls are specified to be of a different type (ι_{τ_2}) than those based upon the target type τ_1. Note also that the "$\Leftrightarrow$" may not be replaced by a "$\Rightarrow$" as in the projective join dependency case. The presence of of an *AB* component unmatched by a *BC* component is represented by a tuple of the form (a,b,η_2); in this case (a,b,ν_{τ_1}) will not be in the database.

3.1.5 Null limiting constraints In the traditional case without nulls, the enforcement of a join dependency is sufficient to guarantee a decomposition into independent components. However, in our framework this is not the case. The unbridled use of nulls can destroy the integrity of a decomposition. The constraints which are necessary to ensure that a decomposition are a generalization (to the typed setting) of the *disjunctive existence constraints* of Goldstein [Gold81]. For $W=(Z,s)\in\mathbf{U}\times\mathsf{Simple}(\mathcal{T},n)$ and $\mathbf{Y}$ a set of subsets of same, we write $\mathsf{NullFill}(W\Rightarrow\mathbf{Y})$ to denote the constraint which states that whenever a tuple *u* exists in the database of type $\hat{s}$ with exactly the *Z* entries nonnull, there exists $(X,v)\in\mathbf{Y}$ and a tuple *t* in the database such that $\pi\langle X\rangle\circ\rho\langle v\rangle(t)=t$ and $t\leq u$. Define NullSat(J) to be $\{\mathsf{NullFill}(W\Rightarrow T)\mid T\in\mathsf{Objects}(J)\}$. We then have the following.

3.1.6 Theorem – main decomposition *Let $J=\bowtie[X_1\langle t_1\rangle, X_2\langle t_2\rangle,..,X_k\langle t_k\rangle]\langle t\rangle$ be a bidimensional join dependency. Then $\{[\pi\langle X_i\rangle\circ\rho\langle t_i\rangle]_\dagger\mid 1\leq i\leq k\}$ is a decomposition of the view defined by $\pi\langle X\rangle\circ\rho\langle t\rangle$ iff the following constraints are met.*

(i) $\mathsf{Con}(\mathbf{D})\models J$.

(ii) $\mathsf{Con}(\mathbf{D})\models\mathsf{NullSat}(J)$.

(iii) *Let α_i denote the constraints implied by $\mathsf{Con}(\mathbf{D})$ on $\pi\langle X_i\rangle\circ\rho\langle t_i\rangle$. Then $\{\bowtie[X_1\langle t_1\rangle, X_2\langle t_2\rangle,..,X_k\langle t_k\rangle]\langle t\rangle\}\cup\mathsf{NullSat}(J)\cup\mathsf{Aug}(\mathbf{A})\cup(\bigcup_{i=1}^k\alpha_i)\models\mathsf{Con}(\mathbf{D})$.* □

Conditions (i) and (ii) ensure representability, while (iii) guarantees independence. The latter condition has been termed *embedding a cover* in the case of projections on schemata constrained by functional dependencies [GrYa84]. In our example of the decomposition based upon $\bowtie[AB,BC,CD,DE]$, the dependency $\bowtie[ABC,CDE]$, although a consequence of $\bowtie[AB,BC,CD,DE]$, does not itself decompose the database. In fact, it is condition (ii) above which is violated. If we only project on *ABC* and *CDE*, we lose those tuples with only two components non-null.

3.2 Simplicity of Decomposition

Given that bidimensional join dependencies generalize ordinary join dependencies, it is natural to ask if the results on acyclicity [BFMY83] may be extended to include them. The answer is generally "yes" for properties which are characterized operationally, such as the existence of a full-reducer, a monotone join plans, or semantic equivalence to a set of multivalued dependencies. The

extension of these results to the graph and hypergraph characterized properties, such as acyclicity, join tree characterization, and the like, is much more involved, and will not be presented here.

Since the definitions here are direct extensions of those of the classical theory (which may be found in [BFMY83] and [Maie83]), we will not elaborate upon them, but only highlight those aspects which indicate how they are extended to the more general framework of this paper.

3.2.1 Definition Let $J = \bowtie[X_1\langle t_1\rangle, X_2\langle t_2\rangle, .., X_k\langle t_k\rangle]\langle t\rangle$ be a bidimensional join dependency, and let $I \subseteq \{1, .., k\}$.

(a) The *I-join dependency* with respect to J is the join dependency $\bowtie[\{\mathsf{Objects}(X_i\langle t_i\rangle) \mid i \in I\}]$. The I-join is the view defined by this dependency. More formally, in the formula $(*)$ of 3.1.1, do the following.

(i) Delete $\Lambda(X_i, t_I)$ for each $i \notin I$.

(ii) For each $j \notin \bigcup_{i\in I} X_i$, replace β_i with $x = \iota_{\tau_j}$.

(iii) For each $i \in I$, replace the R in $\Lambda(X_i, t_i)$ with R_i.

Now just regard the left-hand formula as the defining formula for a view, with R_i identifying the relation of the i^{th} view of the join. The result is recaptured in $R(x_1, x_2, .., x_n)$. We denote this view by $\mathsf{CJoin}(I, J)$. If $I = \{i\}$ and/or $j = \{j\}$, we may write $\mathsf{CJoin}(i, j)$. A state $W \in \mathsf{DB}(\mathbf{D})$ is *join minimal* for J if for any other state W' such that W and W' have the same image under $\mathsf{CJoin}(\{1, .., k\}, J)$, $W \subseteq W'$.

(b) The *I-semijoin* with respect to $j \in I$ is the result of applying the operator $\sum_{i\in I\setminus\{j\}} \pi\langle X_i\rangle \circ \rho\langle t_i\rangle$ to the result of $\mathsf{CJoin}(I, J)$. This semijoin is denoted $I \rhd\!\lhd j$, with the same convention for singletons as described for the join.

3.2.2 Definition Let $J = \bowtie[X_1\langle t_1\rangle, X_2\langle t_2\rangle, .., X_k\langle t_k\rangle]\langle t\rangle$ be a bidimensional join dependency over D.

(a) A *semijoin program* is defined analogously to that of the traditional case. Namely, it is a seqence of pairs $\Theta = \{(\phi_1, \psi_1), (\phi_2, \psi_2), .., (\phi_\ell, \psi_\ell)\}$ in $\{1, .., k\}^2$. We start out with a state W of the base schema $\mathbf{D}$. To apply pair (ϕ_i, ψ_i), we compute the components $\pi\langle X_{\phi_i}\rangle \circ \rho\langle t_{\phi_i}\rangle(W)$ and $\pi\langle X_{\psi_i}\rangle \circ \rho\langle t_{\psi_i}\rangle(W)$, compute the semijoin $\phi_i \rhd\!\lhd \psi_j$, and replace, in W, $\pi\langle X_{\phi_i}\rangle \circ \rho\langle t_{\phi_i}\rangle(W)$ with the result of this semijoin. We just apply the pairs in sequence. It is clear that this will reduce the size of W, hence the term reducer. Θ is a *full reducer* for J if there is a semijoin program which reduces W to a join minimal one.

(b) A *sequential join expression* is represented as a permutation ζ of the set $\{1, .., k\}$. We first compute $\mathsf{CJoin}(\{\zeta(1)\}, J)$, then $\mathsf{CJoin}(\{\zeta(1), \zeta(2)\}, J)$, and so on, until we have computed $\mathsf{CJoin}(\{1, ..k\}, J)$. The expression is *monotone* if each computation yields a view with more tuples than the previous one.

(c) A *tree join expression* (or *join plan*) is a binary tree whose leaves are members of $\{1, .., k\}$. We proceed as above, except that joins may be computed in any order permitted by the tree. The expression is *monotone*, as above, if each computation yields a view with more tuples than the previous one.

We may now state our main structural theorem.

3.2.3 Theorem – characterization of simplicity *Let* $J = \bowtie[X_1\langle t_1\rangle, X_2\langle t_2\rangle, .., X_k\langle t_k\rangle]\langle t\rangle$ *be a bidimensional join dependency on* **D**. *Then the following conditions are equivalent.*

(i) *J has a full reducer.*

(ii) *J has a monotone sequential join expression.*

(iii) *J has a monotone join expression.*

(iv) *J is semantically equivalent to a set of bidirectional multivalued dependencies.* □

4. Conclusions and Further Directions

4.1 Conclusions

We have presented a the basic features of a multi-layered theory of database schema decomposition. First of all, we identified a general algebraic framework which enabled us to state precisely what is meant by a decomposition of a schema into independent components. Such a formulation is particularly important because it places definitions of concepts such as view independence on a firm conceptual ground, unbound by the nuances of particular decomposition frameworks.

Our next step was to build upon this general algebraic framework a subframework capable of dealing with an important spectrum of database decomposition modelling issues. We showed that a type-based framework is useful not only in its own right as a modelling tool for domain objects with structure, but also that a formal theory of null values may be embedded in such a framework, with nulls regarded as special types with special properties. In particular, it was shown to be possible to view the database operation of projection as a special case of restriction.

Our final step was to investigate decomposition within this framework. The notion of a bidimensional join dependency, allowing decomposition in both the horizontal and vertical direction and thus generalizing ordinary join dependencies, was formulated. The notion of acyclicity for ordinary join dependencies was also generalized to this framework, although in an operational rather than hypergraph-theoretic form.

4.2 Further Directions

There are several key directions suggest by our initial investigations. First of all, our formulation of bidimensional join dependencies in general, and vertical join dependencies in particular, was in the presence of null values. While many other researchers have formulated theories of nulls, we know of no other which explicitly investigates the behavior and inference rules of join dependencies in the presence of nulls. Our initial investigations show that all of the usual rules of inference for join dependencies do *not* hold in the presence of nulls. Since nulls seem to be a necessary entity in join decompositions, and join dependencies are ubiquituous in database decomposition, an investigation into the interaction of nulls and inference rules for join dependencies seems warranted.

A second route of further investigation is the generalization of acyclicity. While it is not difficult to extend the operational equivalents of acyclicity to the bidimensional framework (as we have done), the hypergraph-theoretic ones provide more of an obstacle. It is not quite clear what is the meaningful definition of the hypergraph of a bidimensional join dependency. One avenue possibly worth pursuing is that of transforming a bidimensional join dependency into an ordinary join dependency on a larger schema in such a way that the important properties are preserved.

A final route of further investigation is the characterization of decompositions, both ultimate and otherwise. It is clear that not all interesting decompositions are governed by join dependencies. Indeed, the horizontal "split" decompositions proposed by Smith [Smit78] and utilized significantly in distributed database technology are quite possible within this framework. We have not elaborated upon them here because they are, by themselves, rather uninteresting mathematically. However, a general theory of decomposition which admits both bidimensional join based and such split decompositions is well worth pursuing. In particular, we may ask if these two classes of decompositions are complete in the sense that every schema in a certain class has a canonical decomposition into components based upon them.

References

[AhBU79] Aho, A. V., C. Beeri, and J. D. Ullman, "The theory of joins in relational databases," *ACM TODS*, **4**,3(1979), pp. 297-314.

[AïNa86] Aït-Kaci, H., and R. Nasr, "Logic and Inheritance," *Proc. Thirteenth Annual ACM Symposium on Principles of Programming Languages*, 1986, pp. 219-228.

[BaSp81a] Bancilhon, F., and N. Spyratos, "Independent components of databases," *Proc. Seventh VLDB Conf.*, 1981, pp. 398-408.

[BaSp81b] Bancilhon, F., and N. Spyratos, "Update semantics of relational views," *ACM Transactions on Database Systems*, **6**(1981), pp. 557-575.

[BeVa81] Beeri, C., and M. Vardi, "On the properties of join dependencies," in *Advances in Data Base Theory*, edited by H. Gallaire, J. Minker, and J. M. Nicolas, pp. 25-71.

[BFMY83] Beeri, C., R. Fagin, D. Maier, and M. Yannakakis, "On the desirability of acyclic database schemes," *JACM*, **30**,3(1983), pp. 479-513.

[Birk67] Birkhoff, G., *Lattice Theory, third edition*, American Mathematical Society, 1967.

[Bisk81] Biskup, J., "A formal approach to null values in database relations," in *Advances in Data Base Theory*, edited by H. Gallaire, J. Minker, and J. M. Nicolas, pp. 25-71.

[BrMy86] Brodie, M. L., and J. Mylopoulos (editors), *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*, Springer-Verlag, 1986.

[ChMe87] Chan, E. P. F., and A. O. Mendelzon, "Independent and separable database schemes," *SIAM J. Computing*, **16**,5(1987), pp. 841-851.

[DGKG86] DeWitt, D. J., G. Grafe, K. B. Kumar, R. H. Gerber, M. L. Heytens, and M. Muralikrishna, "GAMMA – a high performance dataflow database machine," *Proc. Twelfth VLDB Conf.*, 1986, pp. 228-237.

[Fagi82] Fagin, R., "Horn clauses and database dependencies," *JACM*, **29**,4(1982), pp. 952-985.

[Gold81] Goldstein, B. S., "Constraints on null values in relational databases," *Proc. Seventh VLDB Conf.*, 1981, pp. 101-110.

[Grät78] Grätzer, G., *General Lattice Theory*, Academic Press, 1978.

[GrYa84] Graham, M. H., and M. Yannakakis, "Independent database schemas," *J. Comput. System Science*, **28**(1984), pp. 121-141.

[Hegn83] Hegner, S. J., "Algebraic aspects of relational database decomposition," *Proc. Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1983, pp. 400-413.

[Hegn84] Hegner, S. J., "Canonical view update support through Boolean algebras of components," *Proc. Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1984, pp. 163-172.

[Hegn88] Hegner, S. J., Relational Database Decomposition: Logical and Algebraic Foundations, monograph in preparation, 1988.

[Kers86] Kershberg, L. (editor), *Expert Database Systems: Proceedings from the First International Workshop*, Benjamin-Cummings, 1986.

[Maie83] Maier, D., *The Theory of Relational Databases*, Computer Science Press, 1983.

[McMi79] McSkimin, J. R., and J. Minker, "A predicate calculus based semantic network for deductive searching," in *Associative Networks*, edited by N. V. Findler, pp. 205-238.

[Mink87] Minker, J. (editor), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, 1987.

[Ore42] Ore, O., "Theory of equivalence relations," *Duke Math. J.*, **9**(1942), pp. 573-627.

[Reit80] Reiter, R., "Equality and domain closure for first-order databases," *JACM*, **27**,2(1980), pp. 235-249.

[Reit84] Reiter, R., “Towards a logical reconstruction of relational database theory,” in *On Conceptual Modelling*, edited by M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, pp. 191-233.

[Riss77] Rissanen, J., “Independent components of relations,” *ACM TODS*, **2**,4(1977), pp. 317-325.

[Scio80] Sciore, E., The Universal Instance and Database Design, TR #271, Princeton University, 1980.

[Scio82] Sciore, E., “A complete axiomatization of full join dependencies,” *JACM*, **29**,2(1982), pp. 373-393.

[Smit78] Smith, J. M., “A normal form for abstract syntax,” *Proc. Fourth VLDB Conf.*, 1978, pp. 156-162.

[Vard82] Vardi, M. Y., “On decomposition of relational databases,” *Proc. 23rd Annual IEEE FOCS Symp.*, 1982, pp. 176-185.