Journal of Computer and System Sciences **58**, 512–534 (1999)
Article ID jcss.1999.1628, available online at http://www.idealibrary.com on IDEAL®

Equivalence of Keyed Relational Schemas by Conjunctive Queries*

Joseph Albert

Computer Science Department, Portland State University, P.O. Box 751, Portland, Oregon 97207-0751
E-mail: jalbert@acm.org

Yannis Ioannidis[†] and Raghu Ramakrishnan

Computer Sciences Department, University of Wisconsin, 1210 W. Dayton Street, Madison, Wisconsin 53706

Received November 25, 1997; revised November 2, 1998

The concept of two schemas being equivalent is fundamental to database design, schema integration, and data model translation. An important notion of schema equivalence, *query equivalence*, was introduced by Atzeni *et al.*, and used to evaluate the correctness of schema transformations. The logically equivalent notion of *calculous equivalence*, as well as three progressively more general notions of schema equivalence were introduced in 1984 by Hull, who showed that two schemas with no dependencies are equivalent (under all four notions of equivalence) if and only if they are identical (up to renaming and re-ordering of attributes and relations). Hull also conjectured that the same result holds for schemas with primary keys. In this work, we resolve the conjecture in the affirmative for the case of query equivalence based on mappings using conjunctive relational queries with equality selections. © 1999 Academic Press

1. INTRODUCTION

A fundamental concept in database theory is that of *schema equivalence*. Informally, two schemas are *equivalent* if each one can simulate the other in terms of capacity to store database instances and support queries. A related notion is that of *schema isomorphism*. Two schemas are *isomorphic* if they are identical, up to renaming and re-ordering of attributes and relations. An understanding of schema equivalence is important for schema integration in heterogeneous multidatabase systems [4, 18], where two schemas with dependencies describing the semantics of

* This is a Massive Digital Data Systems (MDDS) project sponsored by the Advanced Research and Development Committee of the Community Management Staff. An extended abstract of the present paper appeared in [1].

[†] Present address: Department of Informatics, University of Athens, Hellas, Greece.

the data are given, and one would like to integrate the schemas. Because the schemas to be integrated may have semantic incompatibilities, it may be necessary to transform one or both of the schemas to equivalent schemas in preparation for integration.

For example, consider the following two relational schemas with key dependencies and referential integrity constraints, and suppose one wants to integrate them. Key attributes are underlined, and referential integrity constraints are shown using standard inclusion dependency notation.

student($\underline{ss}$, name, address)	student($\underline{ssn}$, name)
student_record($\underline{ss}$, gpa, advisor)	student_rec($\underline{ssn}$, gpa, address, advisor)
student[ss] $\subseteq$ student_record[ss]	student[ssn] $\subseteq$ student_rec[ssn]
student_record[ss] $\subseteq$ student[ss]	student_rec[ssn] $\subseteq$ student[ssn]
Schema 1	**Schema 2**

Suppose it is desirable to integrate the two schemas by integrating the student relation in the first schema with the student relation in the second schema to form a unified student relation, and to integrate the student_record relation from the first schema with the student_rec relation in the second schema to form a unified student_record relation. In this case, there is a structural incompatibility due to the address attribute of a student being contained in the student relation in the first schema, whereas it is in the student_rec relation in the second schema, so that a fully general integration of the corresponding relations is not possible.

However, it is straightforward to show that the second schema can be transformed into an equivalent schema in which the incompatibility is removed. Such a schema, Schema 2′, in which the address attribute has been moved to the student relation, is shown here with Schema 1.

student($\underline{ss}$, name, address)	student($\underline{ssn}$, name, address)
student_record($\underline{ss}$, gpa, advisor)	student_rec($\underline{ssn}$, gpa, advisor)
student[ss] $\subseteq$ student_record[ss]	student[ssn] $\subseteq$ student_rec[ssn]
student_record[ss] $\subseteq$ student[ss]	student_rec[ssn] $\subseteq$ student[ssn]
Schema 1	**Schema 2′**

Note that in the absence of the inclusion dependencies specified, Schema 2 and Schema 2′ would not be equivalent. With the dependencies that hold on Schema 2, however, the transformation is equivalence preserving, and the incompability has been removed. The two student relations now can be integrated into a unified relation, as can the student_record and student_rec relations.

We need syntactic characterizations of equivalence of relational schemas with various families of dependencies, such as primary keys, referential integrity constraints, and functional dependencies. In particular, one would like to have a set of transformations for which all schemas equivalent to a given schema can be generated by applying some sequence of transformations from the set. These would form the set of transformations one would implement in an integration tool to provide a complete solution to the problem of restructuring of schemas with the given family of dependencies.

Schema equivalence also is important in database design [5, 8, 10, 19], where given a schema proposed for some application, one may want to choose an equivalent schema that satisfies some desirable normal forms. Indeed, schema equivalence was proposed originally in this context by Codd [8], wherein two schemas are considered equivalent if they support the same queries. Subsequently, a notion of schema equivalence was proposed in which two schemas, both of which are decompositions of the same universal relation, are equivalent if the set of instances of the universal relation for which the decomposition is lossless is the same for both schemas [6, 13]. That is, either schema can *represent* the same set of universal instances. This notion of equivalence is useful for database design, but has the limitation that, as defined, it only applies to pairs of schemas, both of which are projections of the same universal relation scheme. In general, this universal relation assumption is not feasible in multidatabase schema integration, since the schemas are designed and controlled autonomously. Moreover, closed-form characterizations of this form of equivalence are not available, although an algorithm to test for such equivalence is given in [6]. Similar notions of equivalence were defined in [2, 15].

Another notion of equivalence that has been proposed considers two schemas to be equivalent if there is a bijection between the set of database instances of one schema and the set of instances of the other [14, 16, 17]. However, this simply means that the set of instances of one schema has the same cardinality as the set of instances of the other schema. Moreover, if the domain of values available to store in a database is (countably) infinite, then all (nonempty) schemas are equivalent under this notion of equivalence.

The limitations of the notions of equivalence described above are overcome by the notion of query equivalence that was introduced in [3], and studied by Hull, who, in addition, introduced three progressively more general notions of equivalence, Z-generic equivalence, Z-internal equivalence, and absolute equivalence, and provided a rich foundation of theoretical results on schema equivalence [9]. Hull also showed that for relational schemas with no dependencies all four notions of schema equivalence are logically equivalent and that two relational schemas with no dependencies are equivalent if and only if they are isomorphic. Thus a characterization of schema equivalence for relational schemas with no dependencies is available.

Hull also conjectured that this result should generalize to relational schemas with primary keys, that is, that they are equivalent if and only if they are isomorphic. In the present work, we resolve this conjecture in the affirmative for *conjunctive query equivalence*, where instance mappings are conjunctive relational algebra queries with equality selections. (Query equivalence in [9] uses the full relational algebra for such mappings.)

Such a result demonstrates that two schemas whose only dependencies are primary keys support the same conjunctive queries if and only if they are isomorphic. The result also is a negative result about the existence of nontrivial equivalence-preserving transformations for schemas with only primary keys, suggesting that other dependencies are important for transforming schemas in meaningful ways. Indeed, the earlier example shows that when both primary key dependencies and referential integrity constraints are available, there are nontrivial equivalence-preserving schema transformations.

A characterization of equivalence for schemas whose only dependencies are primary keys, however, is critical to obtaining similar characterizations for schemas with other dependencies. For instance, when schemas with primary keys and referential integrity constraints are considered, a schema with only primary keys is a degenerate case where it happens that no referential integrity constraints have been specified. Thus, it would be impossible to characterize equivalence of schemas with primary keys and referential integrity constraints without also characterizing equivalence of schemas with only primary keys.

The remainder of this paper is organized in the following manner. Section 2 presents the formalism used for familiar concepts from the database literature. Section 3 contains the theoretical results contributed by the present work, along with definitions of new concepts presented as they are used. Concluding remarks are made in Section 4.

2. FORMAL DEFINITIONS

2.1. Schemas

In this section, we formalize what is meant by a schema and define various concepts and notation. We assume that the reader is familiar with the relational model of data [7]. A *domain* is a countably infinite set of atomic elements. A collection of *attribute types* over some domain $\mathscr{D}$ is a finite collection of disjoint subsets of $\mathscr{D}$. Attribute types are also (countably) infinite. An *attribute* is a pair consisting of a name (called the name of the attribute) and an attribute type (called the type of the attribute). A *relation scheme* consists of a name (name of the relation) and an ordered list of attributes, generally written $R[A_1, A_2, ..., A_k]$. R is the name of the relation. Note that an attribute belongs only to a single relation.

For each i, if N_i is the type of attribute A_i, then each of the finite subsets of the cross-product $N_1 \times N_2 \times \cdots \times N_k$ is called an *instance* of relation R. The tuple $\langle N_1, N_2, ..., N_k \rangle$ is called the *type* of the relation R. A *relational database schema* is a tuple of relation schemes. A database instance of the schema is a tuple of instances of each relation scheme in the database schema. We write $i(S)$ for the set of all instances of schema S.

A *key dependency* on a relation is a declared subset of the attributes of the relation. The subset of attributes is called a key. A key dependency on some relation is satisfied by an instance of the relation if every pair of distinct tuples in the instance differ in value of at least one of the attributes in the key. A subset of the attributes of some relation is called a *superkey* if it is a superset of a key. A *keyed schema* is one where a single key is specified for each relation in the schema, and no other dependencies are specified to hold in the schema. A schema for which no dependencies are specified is called an *unkeyed* schema.

A *functional dependency* on a schema is a declaration of a pair of attribute sets. If X and Y are the two sets of attributes, the dependency is usually written $X \rightarrow Y$. Sometimes the elements of one or both of the sets are listed explicitly as a string. For instance, if $Y = \{A, B\}$, then the dependency $X \rightarrow Y$ might be written $X \rightarrow AB$. If all of the attributes in both X and Y belong to the same relation, then an instance of that relation is said to satisfy the dependency if every pair of tuples of the relation instance that differ on some attribute in Y also differ on some attribute in X. Otherwise, the dependency fails for the given relation instance. An instance of the schema satisfies some functional dependency $X \rightarrow Y$ if all of the attributes in both X and Y belong to the same relation, and the instance of this relation in the database instance satisfies the dependency. If the attributes in X and Y fail to belong to the same relation, then the functional dependency fails for any instance of the schema. Note that allowing functional dependencies to be expressed in this way differs from the usual formalization where a functional dependency is only defined using attributes from a single relation, but this trivial extension allows for a concise statement of some of the results below.

2.2. *Queries and Query Mappings*

A *view* over a schema S is a pair (V, q), where V is a relation scheme and q: $i(S) \rightarrow i(V)$ maps each instance of S to an instance of V. The mapping q is called a *query*. If $q(d) = a$ for some database d, then a is called the *answer* to the query q for database d. The type of the view and the type of the query are both defined as the type of V. A *query language* consists of a syntax capable of specifying a set of syntactic objects, and an assignment of each syntactic object in the set to a query. The assignment is said to define the semantics of the language. The syntactic objects in the language usually also are referred to as queries.

The notion of a query mapping from one schema to another is an important concept and is defined as follows.

Definition. Given schemas $S_1 = \langle R^1_1, R^1_2, ..., R^1_n \rangle$ and $S_2 = \langle R^2_1, R^2_2, ..., R^2_m \rangle$, and a query language, $\mathscr{L}$, then $\alpha = \langle v_1, v_2, ..., v_m \rangle$ is a *query mapping* from S_1 to S_2 if each v_k is a view over S_1 defined using queries in $\mathscr{L}$, and the type of v_k is the same as the type of R^2_k for each k.

For each instance of S_1, the query mapping defines an instance of S_2, since each v_k defines an instance of R^2_k. We write α: $i(S_1) \rightarrow i(S_2)$. If α is a query mapping from a keyed schema S_1 to a keyed schema S_2, then we say that α is *valid* if it maps each instance of S_1 satisfying the key dependencies for S_1 to an instance of S_2 satisfying the key dependencies for S_2. Query mappings between unkeyed schemas are always valid.

The notion of a conjunctive query is well known in the database literature [12] and can be defined in the following manner.

Definition. A *conjunctive query* is a relational algebra query that can be expressed using only the operations of select, project, and join.

A conjunctive query view (V, q) is specified using a syntactic style borrowed from Datalog [11]. However, the syntax used here is more restrictive than Datalog, allowing only distinct variables as placeholders in columns of relations, with all selection and join conditions occurring in a separate list of equality predicates included in the conjunct. The general form of a query is shown in (1):

$$V(A_1, A_2, ..., A_n) :- R_1(X^1_1, ..., X^1_{i_1}), ..., R_k(X^k_1, ..., X^k_{i_k}), \textit{equality-list}. \qquad (1)$$

Each R_i is a relation, and each X_i^j is a distinct variable serving as a placeholder. The A_i's are not necessarily distinct and are either constants or variables that occur among the X_i^j variables to signify that this constant or variable is in the result of the query. Other variables might be dummy placeholders to signify attributes in some R_i that are projected out of a relation or variables participating in joins or selections whose columns subsequently are projected out of the final result.

As with Datalog, the comma-separated list of relations forms a cross-product to which the conditions in the equality list are applied. The equality list is a list of equality predicates with form either $X = Y$ or $X = a$. In the first case, the two variables X and Y are being equated. If both X and Y are used as placeholders in the same relation, then this is a *column selection*, whereas if the two variables occur in different relations, then this corresponds to a *join condition*. For the equality predicate $X = a$, the column of the relation containing the variable X as a placeholder in the query has a *selection condition*, selecting tuples with value for that attribute equal to the constant a. Constants may occur explicitly among the A_i. All variables occurring in equality predicates in the equality list must also occur as a placeholder for some attribute in some relation occurring in the body of the query. Note that all conjunctive relational algebra queries with equality selections can be expressed with the syntax just described. For the remainder of this paper, "conjunctive query" means "conjunctive query with equality selections."

Note that the equality of variables in the equality list of a conjunctive query induces a natural equivalence relation on the variables. In particular, V_1 is equivalent to V_2 if either $V_1 = V_2$ appears in the equality-list, or can be inferred from the equality list by reflexivity, symmetry, and transitivity. We call the equivalence classes generated by this equivalence relation the *equality classes* of variables. In other words, V_1 and V_2 belong to the same equality class if $V_1 = V_2$ can be inferred from the equality predicates in the equality-list. Each such variable is a placeholder for an attribute in some relation, and for a given equality class of variables that span multiple relations, we say that there is a *join* between the relations in which the placeholder variables occur. In such a case, we also say that the attributes corresponding to the placeholder variables in the equality class *participate* in the join.

The notions of containment and equivalence of queries are well known [12] and can be defined in the following manner.

Definition. Given two queries $q\colon i(S) \to i(V)$ and $q'\colon i(S) \to i(V)$ that have the same type, we say that q is *contained in* q', written $q \sqsubseteq q'$, if for every $d \in i(S)$, $q(d) \subseteq q'(d)$.

Definition. We say that q is *equivalent* to q', written $q \cong q'$, if $q \sqsubseteq q'$ and $q' \sqsubseteq q$.

2.3. Dominance and Equivalence

In this section, we formalize the notions of schema dominance and schema equivalence. Dominance is defined first.

Definition. Let S_1 and S_2 be two keyed schemas, and let $\mathscr{L}$ be a query language. Then we say that S_2 *$\mathscr{L}$-dominates* S_1, written $S_1 \preccurlyeq_{\mathscr{L}} S_2$, if there are valid query maps $\alpha\colon i(S_1) \to i(S_2)$ and $\beta\colon i(S_2) \to i(S_1)$ such that $\beta \circ \alpha$ is the identity map on $i(S_1)$. To indicate the query mappings that establish the dominance we sometimes write $S_1 \preccurlyeq_{\mathscr{L}} S_2$ by (α, β).

Schema equivalence is now defined in terms of schema dominance.

DEFINITION. If $S_1 \preccurlyeq_{\mathscr{L}} S_2$ and $S_2 \preccurlyeq_{\mathscr{L}} S_1$, then we say that the two schemas are *$\mathscr{L}$-equivalent*, written $S_1 \equiv_{\mathscr{L}} S_2$.

These notions of $\mathscr{L}$-dominance and $\mathscr{L}$-equivalence were introduced in [3]. We sometimes write $S_1 \preccurlyeq S_2$, or $S_1 \equiv S_2$, when the particular language $\mathscr{L}$ is clear from the context. The following result is proved in [9].

THEOREM (Hull [9]). *If $\mathscr{L}$ is the relational algebra, and S_1 and S_2 are schemas with no dependencies, then $S_1 \equiv_{\mathscr{L}} S_2$, if and only if S_1 and S_2 are isomorphic.*

Hull also conjectured that this result holds for keyed schemas, but this conjecture remains open.

3. EQUIVALENCE RESULTS

3.1. Overview

The main result of the paper, that keyed schemas are equivalent under query equivalence by conjunctive relational algebra queries with equality selections if and only if they are isomorphic, is given as Theorem 13 in Section 3.5. The proof divides into two phases. The first phase involves demonstrating that if some keyed schema S_1 is dominated by some other keyed schema S_2, then we can delete all of the nonkey attributes from both schemas while preserving dominance. This result, the preservation of dominance by reduction to keys, is given as Theorem 11 in Section 3.3. The importance of this result is that unkeyed schemas result from deleting the nonkey attributes from a keyed schema, enabling results about unkeyed schemas to be used to reason about keyed schemas. In particular, it will be used later in the proof to show that if S_1 and S_2 are equivalent, then there must be a one-to-one correspondence of relations with isomorphic keys between the schemas. That is, the two schemas must agree on the number of relations and the sets of keys, so that any differences in the two schemas only can be in the nonkey attributes.

The second phase of the proof involves reasoning about the nonkey attributes. The central result used for this purpose is Theorem 12 of Section 3.4, that ensures the preservation of functional dependencies across mappings that establish dominance. When some keyed schema S_1 is dominated by another keyed schema S_2 by query maps α and β, values for attributes in the instance of S_1 are encoded in attribute values in some instance of S_2 by α, and these values are mapped back to the instance of S_1 by β. If a functional dependency holds in S_2 among attributes that are used in such a way to encode information from some instance S_1, then the theorem asserts that an analogous functional dependency must hold among the attributes of S_1 that are being encoded in these attributes in S_2. Because the only functional dependencies that hold in a keyed schema are ones where the left-hand side is a superkey, this result can be used to reason about correspondences in the nonkey attributes in equivalent schemas.

For ease of presentation, we will drop the $\mathcal{L}$ for the remainder of the paper, and write $S_1 \preccurlyeq S_2$ by (α, β) to mean that S_2 dominates S_1 by the conjunctive query mappings α and β.

3.2. *Basic Results*

In this section, we present a number of technical lemmas that are used throughout the proof. Definitions of new concepts, such as identity joins, ij-saturated (identity join saturated) queries, and product queries, are given where they are first used. We start with the definition of identity joins.

Definition. In a conjunctive query, a join condition is an *identity join condition* if the variables being equated by the join condition are both placeholders for the same attribute in (different occurrences of) the same relation.

For example, in the query, $Q(X, Y, Z) :- R(X, Z), R(Y, T), Z = T.$, the join condition is an identity join condition.

Definition. In a conjunctive query, a join is an *identity join* if all of the join conditions associated with the join are identity join conditions.

For example, in the query, $Q(X, Y, Z) :- R(X, Y, Z), R(U, V, T), Y = V, Z = T.$, the join is an identity join. This is because the join is of a relation with itself, and the join conditions equate, respectively, the second and third attributes with (another copy of) themselves. On the other hand, in the query, $Q(X, Y, Z) :- R(X, Y, Z), R(T, U, V), Y = T, Z = V.$, there is a self-join that is *not* an identity join. In this case, the join condition $Y = T$ equates two different attributes of relation R. A cross-product of a relation with itself (some number of times) satisfies the definition of an identity join vacuously and, thus, is considered to be a degenerate identity join.

Next, we define the notions of ij-saturated relations and ij-saturated queries.

Definition. A relation R occurring in the body of a conjunctive query is *ij-saturated* in the query if no occurrence of R in the query participates in a selection condition, all join conditions involving R are identity join conditions, and all possible identity join conditions for R can be inferred from the equality conditions specified.

Thus, R is ij-saturated in the query (2):

$$Q(X, Y) :- R(X, Y), R(A, B), R(C, D), X = A, X = C, Y = B, Y = D. \tag{2}$$

The join conditions $A = C$ and $B = D$ are inferred by transitivity. But R is not ij-saturated in the query (3):

$$Q(X, Y) :- R(X, Y), R(A, B), R(C, D), X = A, X = C, A = C, Y = B. \tag{3}$$

This is because neither $Y = D$ nor $B = D$ can be inferred from the equality list.

Definition. A conjunctive query is ij-saturated if every relation that occurs in its body is ij-saturated in the query.

Note that, given any conjunctive query q that has no selection conditions and no join conditions other than identity join conditions, we can construct an ij-saturated query $\hat{q}$ that has the same number of occurrences of relations in its body as the original query q, but with extra identity join conditions added so each relation is ij-saturated. For example, given the query (4), we can construct the ij-saturated query (5):

$$Q(X, Y) :- R(X, Y), R(A, B), R(C, D), X=A, X=C, A=C, Y=B. \tag{4}$$

$$Q'(X, Y) :- R(X, Y), R(A, B), R(C, D), X=A, X=C, A=C, Y=B, Y=D, B=D. \tag{5}$$

Note that $\hat{q} \sqsubseteq q$ always holds because $\hat{q}$ is the result of adding extra join conditions to q.

A conjunctive query is a *product query* if there are no selection or join conditions, and every relation occurring in the body of the query occurs only once. That is, a product query can consist of only a single relation or a cross-product of distinct relations.

The first lemma demonstrates a basic property of ij-saturated queries and product queries.

LEMMA 1. *Every ij-saturated query is equivalent to a product query having the same relations in its body as the ij-saturated query.*

Proof. Let q be an arbitrary ij-saturated query. Construct a query q' as follows:

1. eliminate all (identity) join conditions from the body of q;
2. eliminate all duplicate occurrences of any relation in the body of q.
3. replace any variable X in the head of the query that no longer occurs in the body with a variable Y that does still occur in the body, such that $X=Y$ is an identity join condition that can be inferred from the join conditions specified in the ij-saturated query. Such a Y always exists because q is ij-saturated.

Clearly the resulting query q' is a product query having the same relations in its body as q. We claim that $q \cong q'$. By construction, q and q' both have the same type and are defined over the same schema. Let this schema be S and let d be an arbitrary instance of S. To show that $q \sqsubseteq q'$, let $\tau \in q(d)$. Then there must be tuples in each relation occurring in the body of q that are used in inferring τ. But then, since each relation in the product query q' occurs in the body of q, and q' has no selections or joins, τ must be a member of $q'(d)$, as required. To show that $q' \sqsubseteq q$, let $\tau' \in q'(d)$. Then there must be tuples in each relation occurring in the body of q' that are used to infer τ'. But since any tuple of a relation always satisifies an identity join on that relation, these tuples will satisfy the identity join conditions. Thus, $\tau' \in q(d)$ as required, establishing the lemma. ▌

Lemma 1 can be used to show what properties can be preserved when identity join conditions are removed from a query that has no selection or join conditions other than identity join conditions. This is the subject of the next lemma.

Lemma 2. *If q is a conjunctive query defined on some schema S and q has no selection conditions nor any join conditions that are not identity join conditions, then there exists a product query $\hat{q}$ satisfying the conditions*:

(a) $\hat{q} \sqsubseteq q$;

(b) *for every $d \in i(S)$, any functional dependency that holds on $q(d)$ also holds on $\hat{q}(d)$*;

(c) *for every $d \in i(S)$, if $q(d)$ is nonempty, then $\hat{q}(d)$ is nonempty*;

(d) *all of the relations occurring in the body of q also occur in the body of $\hat{q}$.*

Proof. Let q be an arbitrary conjunctive query having no selection conditions nor any join conditions that are not identity join conditions. Then let q' be the ij-saturated query that results from adding to q all the valid identity join conditions missing from q. Clearly $q' \sqsubseteq q$, since q' results from adding join conditions to q. If we take $\hat{q}$ to be the product query that Lemma 1 guarantees to be equivalent to q', then $\hat{q}$ is a product query satisfying conditions (a) and (d) of the lemma. Condition (b) follows immediately from the fact that $\hat{q} \sqsubseteq q$, since if some functional dependency fails on $\hat{q}(d)$ for some database state d, then there is a pair of tuples in $\hat{q}(d)$ that violate the dependency, and these tuples are in $q(d)$ also, whence the functional dependency fails on $q(d)$. For condition (c), let $\hat{q}$ be empty when evaluated over some database instance d. Then, since $\hat{q}$ is a cross product of the relations occurring in q (along with a projection to the attributes occurring in the head of q), it follows that one of these relations is empty in the database instance d. But then, the query q is empty when evaluated over d as well, establishing condition (c), and the lemma follows. ▌

Throughout many of the proofs below, it will be necessary to reason about the manner in which specific data in one database instance is encoded in another instance. We define the following syntactic concept for conjunctive queries to capture an essential property of the encodings.

Definition. For any attribute A assigned from a column in the result of a conjunctive query, we say that A *receives* attribute B from relation R if, in the representation of the query, A is assigned from a variable that occurs at, or is equated to a variable at, the location of attribute B in R. If an attribute A is assigned by a constant symbol, then we say that attribute A receives the constant.

Thus, in the query, $R(X, Y, Z) :- P(X, Y), Q(T, Z),\ Y = T.$, the second attribute of relation R receives from P the second attribute listed in the scheme of P, and it also receives from Q the first attribute listed in the scheme of Q. Similarly, in the query: $R(a, Y, X) :- P(X, Y).$, the first attribute of relation R receives the constant a. An attribute can receive multiple, distinct attributes, as shown in the first example, but an attribute never receives both an attribute and a constant.

A very useful notion, *attribute-specificity*, can be used to simplify various arguments and is used in several of the proofs below. This is now defined.

Definition. A database instance d of some schema is *attribute-specific* if for any two distinct attributes A and B, $\pi_A(d) \cap \pi_B(d) = \emptyset$.

The notion of attribute-specific database instances will be used to derive various properties of conjunctive query mappings. The following lemma can be used to demonstrate the existence of attribute-specific database instances that satisfy certain properties.

Lemma 3. *Given some keyed schema S and some finite set of domain elements F, then there exists a valid attribute-specific instance of S such that all relation instances are both nonempty and contain no elements of F. Moreover, if there is some functional dependency which fails on some valid instance of the schema, then there exists a valid attribute-specific instance of S such that all relation instances are both nonempty and contain no elements of F, and for which the functional dependency fails.*

Proof. Let S be an arbitrary keyed schema and F be some finite set of domain elements. For each attribute occurring in S, choose a unique element from the domain of the attribute that does not occur in F. Since all attribute types are infinite, this always will be possible. Let d be the database instance of S that has one tuple in each relation instance with the value of each attribute being the unique element that was chosen for the attribute. Then d is a valid attribute-specific instance of S such that all relation instances are both nonempty and contain no elements of F.

For the second part of the lemma, suppose there is some functional dependency that fails on some valid instance of the schema. If the functional dependency references attributes in more than one relation in the schema, then the functional dependency fails for all instances of S and d is a an attribute-specific instance on which the functional dependency fails, as required. On the other hand, if the functional dependency only references attributes that occur in some relation R in S, then because there is a valid instance of S for which the dependency fails, the left-hand side of the dependency cannot be a superkey. For each attribute on the right-hand side of the dependency that does not also occur on the left hand side, choose another unique value from the type of the attribute that does not occur in the set F or in the database instance d. Note that there always will be at least one such attribute on the right-hand side of the dependency that does not occur on the left-hand side, for otherwise, the dependency would be tautological, contradicting that it fails on some valid database instance.

If τ is the tuple in the instance of R in d, then construct another tuple τ' that has the newly chosen value for the corresponding attribute type as the value of some attribute that occurs on the right-hand side of the dependency, but that does not occur on the left-hand side. Let the remaining attributes in τ' have the same value that they have in τ. Let d' be the instance of S that has the same relation instance as d for every relation other than R, but the instance of R in d' contains exactly two tuples, τ and τ'. Because the left-hand side of the functional dependency is not a superkey for R, the database instance d' is valid and is an attribute-specific instance of S for which all relations are both nonempty and contain no elements of F, and for which the functional dependency fails. ∎

Note that the existence of attribute-specific instances depends on the fact that there are no inclusion dependencies, such as referential integrity constraints, allowed in the schemas. If such dependencies were present in some schema, attribute-specific instances of the schema generally would not exist.

The next five lemmas present some properties of attribute encodings in conjunctive query maps that establish dominance. The first of these shows that if $S_1 \preccurlyeq S_2$ then all attributes in S_1 are encoded somewhere in S_2.

LEMMA 4. *If $S_1 \preccurlyeq S_2$ by (α, β) then for every attribute A occurring in S_1 there is some attribute B in S_2 such that A is received by B under α, and B is received by A under β.*

Proof. Let $S_1 \preccurlyeq S_2$ by (α, β), and let A be an arbitrary attribute in schema S_1. Let d be some attribute-specific instance of S_1 that contains for attribute A some value a that is not among any constants in any of the queries in the maps α or β. By Lemma 3, such a d always exists. Since $\beta \circ \alpha$ is the identity map on $i(S_1)$, $\beta(\alpha(d)) = d$. Thus, the value a occurs as a value of attribute A in $\beta(\alpha(d))$, and since a does not occur among any of the constants in the queries in α or β, A must have received some attribute B of S_2 under β. Moreover, B must contain the value a in the instance $\alpha(d)$. Since d is an attribute-specific instance and a does not occur in any query constants, B must receive A under α, establishing the lemma. ▮

The next lemma maintains that if $S_1 \preccurlyeq S_2$ and some attribute in S_2 is mapped back to some attribute in S_1, then the attribute in S_1 is encoded in this attribute in S_2.

LEMMA 5. *If $S_1 \preccurlyeq S_2$ by (α, β) and B is an attribute in S_2, then if B is received by some attribute A in S_1 under β, then A must be received by attribute B under α.*

Proof. Suppose not. Then there is some attribute A in S_1 that receives attribute B under β, but A is not received by B under α. Let d be an attribute-specific database instance having some value a for attribute A that is not among any of the query constants in α or β, as Lemma 3 guarantees will always exist. Since A is not received by B under α, the value a will not be among the values found for attribute B in the database instance $\alpha(d)$. Thus, since A receives attribute B under β, the value a cannot be a value for attribute A in the database instance $\beta(\alpha(d))$. This is because either A receives attribute B and no other attributes under β, in which case the absence of a from the values for B ensure that a is not a value for A, or A receives attribute B as well as one or more other attributes under β. In the latter case, all the attributes received by A under β have placeholder variables that are in the same equality class, and the absence of a from the values for B ensures that a is not a value for A in $\beta(\alpha(d))$. But since a is a value for A in d, this contradicts that $\beta \circ \alpha$ is the identity map on $i(S_1)$. ▮

The following lemma ensures that if $S_1 \preccurlyeq S_2$, then two different attributes in S_1 cannot be encoded in the same attribute in S_2.

LEMMA 6. *Let S_1 and S_2 be keyed schemas such that $S_1 \preccurlyeq S_2$ by (α, β). Then there cannot be two distinct attributes in S_1 that receive the same attribute in S_2 under β.*

Proof. Let S_1 and S_2 be as in the statement of the lemma, and suppose to the contrary that there is some attribute B in relation R in S_2 that is received under β by two distinct attributes A and A' in S_1. By Lemma 5, B receives both A and A' under α. By Lemma 3, there exists an attribute-specific instance d of S_1 such that the relation(s) containing A and A' is/are nonempty. Since A and A' are distinct attributes, they have no values in common in d. But B receives both A and A' under α, which means that there is a join condition between A and A' in the query defining R. Because d is attribute-specific, the join condition will fail, so that R must be empty, whence the relation containing A must be empty in the instance $\beta(\alpha(d))$, since A receives B under β. This is a contradiction, establishing the lemma. ▮

The next two lemmas demonstrate that if $S_1 \preccurlyeq S_2$ and every attribute type occurs in both schemas the same number of times, then every attribute in S_2 is used to encode something in S_1 in an essential way and that there are no extraneous encodings of attributes from S_1 in S_2.

LEMMA 7. *Let S_1 and S_2 be keyed schemas such that $S_1 \preccurlyeq S_2$ by (α, β). If, for every attribute type T, the number of attributes in S_1 of type T is the same as the number of attributes in S_2 of type T, then every attribute in S_2 is received by some attribute in S_1 under β.*

Proof. Suppose not. Then there is some attribute C in S_2 which is not received by any attribute in S_1 under β. By Lemma 4, for every attribute A in S_1, there is some attribute B such that A is received by B under α and B is received by A under β. Since, in a query mapping, any attribute of some type T that is received by another attribute is always received by an attribute with the same type T, and since both schemas have the same number of occurrences of each attribute type, it follows that if T_C is the type of attribute C, there must be fewer attributes of type T_C in S_2 that are received by attributes of type T_C in S_1 than there are attributes of type T_C in S_1. It follows that there must be two distinct attributes A and A' in S_1 such that the same attribute B is received by both A and A' under β, contradicting Lemma 6. ▮

LEMMA 8. *Let S_1 and S_2 be keyed schemas such that $S_1 \preccurlyeq S_2$ by (α, β). If, for every attribute type T, the number of attributes in S_1 of type T is the same as the number of attributes in S_2 of type T, then there cannot be two distinct attributes in S_2 that are received by the same attribute in S_1 under β.*

Proof. Let S_1 and S_2 be as in the statement of the lemma and suppose to the contrary that some attribute A in S_1 receives two distinct attributes from S_2 under β. Then, since Lemma 7 guarantees that every attribute in S_2 is received by some attribute in S_1 and since the number of attributes of each type is the same in both schemas, it follows by simple counting that there must be some other attribute in S_2 that is received by two or more distinct attributes in S_1, contradicting Lemma 6. ▮

3.3. Preservation of Dominance by Reduction to Keys

In this section we prove a theorem that establishes an important property of conjunctive query dominance. In particular, it shows that for one schema to be dominated by a second schema, its key set must be dominated by the key set of the second schema. That is, if we delete the nonkey attributes from both schemas, the same dominance relationship will still hold. The following notation is useful for describing the schemas and instances with nonkey attributes removed.

Definition. If S is a keyed schema, $\kappa(S)$ is the unkeyed schema that is obtained by deleting all nonkey attributes from each relation scheme in S and dropping the key dependencies. Thus, for each relation scheme R in S, there is a relation scheme R' in $\kappa(S)$ whose scheme consists only of the key attributes of R.

Definition. If S is a keyed schema and d is a database instance of S, then $\pi_\kappa(d)$ is the database instance of $\kappa(S)$ that corresponds to projecting all of the nonkey attributes out of the database instance d.

The theorem to be proved is important because, given some keyed schema S, $\kappa(S)$ is an unkeyed schema, so this result allows results concerning unkeyed schemas to be used in reasoning about keyed schemas. For instance, if one wanted to show that some keyed schema S_1 were *not* dominated by some other keyed schema S_2, it would suffice to show that $\kappa(S_1)$ was not dominated by $\kappa(S_2)$, which in turn might be demonstrated using techniques concerning unkeyed schemas.

The technique of the proof is to construct the query mappings that establish dominance of some $\kappa(S_1)$ by some $\kappa(S_2)$. The idea is that, given an instance of $\kappa(S_1)$, arbitrary values can be created for the nonkey attributes of S_1 to yield an instance of S_1. Then the dominance mapping that encodes instances of S_1 as instances of S_2 can be applied to get an instance of S_2, and the nonkey attributes can be projected out to get an instance of $\kappa(S_2)$.

To be able to construct a mapping from $i(\kappa(S_2))$ back to $i(\kappa(S_1))$ that recovers the original instance, it is required that when $S_1 \preccurlyeq S_2$ by (α, β), all of the data values for the key attributes in S_1 are encoded in key attributes in S_2, even if they also may be mapped to nonkey attributes by α, and recovered from these nonkey attributes by β. This is established by the following lemma.

Lemma 9. *If S_1 and S_2 are keyed schemas and $S_1 \preccurlyeq S_2$ by (α, β), then if some nonkey attribute B in some relation in S_2 receives some key attribute K in some relation in S_1 under α, and either B is received by K under β, or B is involved in a join or selection condition in the body of some query in β, then*:

(a) *K is received by some key attribute K' in S_2 under α with K' in the same relation as B*; *and*

(b) *for any database instance in the range of α, K' and B have the same value in each tuple of the relation containing them.*

Proof. Let $S_1 \preccurlyeq S_2$ by (α, β) and let K and B be as in the statement of the lemma. Let R_1 be the relation in S_1 containing K, and let R_2 be the relation in S_2 containing B. Let P be a relation in S_1 that is defined by a query in β in which B is used either in a join or selection condition, or B is received by K (P will be the same relation as R_1 if B is received by K under β). Let $\{k_1, k_2\}$be a set of two unique values belonging to the attribute type of attribute K such that neither k_1 nor

k_2 occur as constants in the queries in α or β. Also define a function g on the domain of schema S_1 so that $g(k_1)=k_2$ and $g(k_2)=k_1$, but $g(x)=x$ when $x \neq k_i$ for $i=1, 2$.

Now let d be an attribute-specific instance of S_1 such that all relations of the schema are nonempty, none of the values in d occur as constants in any of the queries in α or β, each attribute other than K has only a single value stored in d, but there are exactly two values, k_1 and k_2 stored for attribute K. Thus, the instance of relation R_1 in d has two tuples, and all other relation instances in d have a single tuple. Such a database instance can be constructed just as was done in the proof of Lemma 3, but an extra tuple is constructed and added into the instance of relation R. Let r_2 be the instance of R_2 in $\alpha(d)$.

First we claim that r_2 is nonempty. In proof, suppose not. Then, since the relation P is defined by a query in β in which B either is received by an attribute in P or B is involved in a join or selection condition, it must be that the instance of P in $\beta(\alpha(d))$ is empty. This is a contradiction because all relations in d are nonempty and $d=\beta(\alpha(d))$, establishing the claim.

Next note that the only selection or join conditions in the query in α that defines R_2 must equate an attribute with itself (possibly in a different occurrence of the same relation in the body of the query). This is because d is an attribute-specific instance containing no query constants, so if there were a selection condition equating an attribute to a constant or a selection or join condition equating two different attributes in the query, r_2 would be empty, a contradiction.

Since r_2 is nonempty, there is some tuple $\tau \in r_2$. Because the instance r_2 is defined by a query in α, there is a derivation of the tuple τ by the query. Let V be the variable in the position of attribute B in the head of the query, and suppose that the body of the query has n relations in it (the occurrences of relations in the body need not be distinct). Let these relations be called Q_i for $i=1, 2, ..., n$. Note that R_1 occurs one or more times among the Q_i. Then for some n, there exist n tuples, $\tau_1, \tau_2, ..., \tau_n$, that are used to derive τ in r_2, and for each i, $\tau_i \in Q_i$. Now, define tuples τ'_i for $i=1, 2, ..., n$ as

$$\begin{array}{ll} \tau'_i.A = g(\tau_i.A), & \text{if } Q_i \text{ is an occurrence of relation } R_1 \text{ and has in the position of} \\ & \text{attribute } K\text{, a variable in the same equality class as } V; \\ \tau'_i.A = \tau_i.A & \text{otherwise.} \end{array}$$

Thus, each $\tau'_i \in Q_i$, and since the only join or selection conditions in the query are ones that equate an attribute with itself, there must be a derivation of some tuple $\tau' \in r_2$ using the tuples τ'_i.

By construction, $\tau'.B = g(\tau.B) \neq \tau.B$, since K is received by B under α. Since B is a nonkey attribute, and two tuples that disagree on a nonkey attribute must also disagree on some key attribute, there must be at least one key attribute of R_2 for which τ and τ' disagree. Let this key attribute be K'. Then it must be that $\tau'.K' = g(\tau.K') \neq \tau.K'$, and by construction, the head of the query that defines R_2 has either variable V, or some other variable in the same equality class as V in the position of attribute K'. But this means that K' receives attribute K under α, establishing part (a) of the lemma, and since the variables in the head of the query in

$$\begin{tikzcd}
S_1 \arrow[r, shift left, "\alpha"] \arrow[d, shift left, "\pi_\kappa"] & S_2 \arrow[l, shift left, "\beta"] \arrow[d, shift left, "\pi_\kappa"] \\
\kappa(S_1) \arrow[u, shift left, "\gamma"] \arrow[r, shift left, "\alpha_\kappa"] & \kappa(S_2) \arrow[u, shift left, "\delta"] \arrow[l, shift left, "\beta_\kappa"]
\end{tikzcd}$$

FIG. 1. Schema mappings.

the positions for attributes K' and B are in the same equality class, K' and B must have the same value in every tuple in any instance of R_2 in the range of α, establishing part (b) of the lemma. ∎

Now we turn to the construction of query maps α_κ and β_κ for which $\kappa(S_1) \preccurlyeq \kappa(S_2)$ by $(\alpha_\kappa, \beta_\kappa)$. If $\mathscr{A}$ is the collection of attribute types and $\mathscr{D}$ the domain of values for the schema S_1, then let $f: \mathscr{A} \to \mathscr{D}$ be some fixed, arbitrary map such that $f(T) \in T$ for each $T \in \mathscr{A}$. That is, the mapping f is a choice function that associates each attribute type with a constant value belonging to that attribute type.

We will define mappings γ and δ so that α_κ is given by $\pi_\kappa \circ \alpha \circ \gamma$, and β_κ is given by $\pi_\kappa \circ \beta \circ \delta$. The mapping relationships are shown in the Fig. 1.

First we define the mapping γ: $i(\kappa(S_1)) \to i(S_1)$ as follows. The mapping γ is a conjunctive query mapping such that for any relation R in S_1 having n key attributes and m nonkey attributes, the instance of R in $\kappa(S_1)$ is defined in γ by the query (6):

$$R(K_1, K_2, ..., K_n, c_1, c_2, ..., c_m) :- R'(K_1, K_2, ..., K_n). \tag{6}$$

Here R' is the relation in $\kappa(S_1)$ corresponding to R but with nonkey attributes projected out. We are assuming without loss of generality that the key attributes of relation R are ordered so that they correspond to the leftmost n variables of R, and that the attributes of R' obey the same order. A similar assumption will be made in the definition of δ below. Each c_i is a constant symbol and $c_i = f(T)$, where T is the type of the attribute corresponding to the position of c_i in R. Note that $\pi_\kappa(\gamma(d_\kappa)) = d_\kappa$, for any database instance d_κ of $\kappa(S_1)$.

Given an arbitrary database instance d of $\kappa(S_1)$, define $\alpha_\kappa(d) = \pi_\kappa(\alpha(\gamma(d)))$. Note that α_κ is a conjunctive query mapping from $\kappa(S_1)$ to $\kappa(S_2)$, since the conjunctive rules for α_κ can be constructed by simple substitution of each (conjunctive) query in γ for the relations appearing in the body of α.

To define the mapping β_κ, we first define the mapping δ: $i(\kappa(S_2)) \to i(S_2)$ in the following manner. The mapping δ is a conjunctive query mapping such that for any relation R in S_2 having n key attributes and m nonkey attributes, the instance of R in $\kappa(S_2)$ is defined in δ by the query (7):

$$R(K_1, K_2, ..., K_n, t_1, t_2, ..., t_m) :- R'(K_1, K_2, ..., K_n). \tag{7}$$

Here R' is the relation in $\kappa(S_2)$ corresponding to R but with nonkey attributes projected out. Each t_i either is a constant symbol or variable and is defined as follows: For each i, let attribute B_i be the attribute of relation R whose placeholder in the conjunctive query is t_i, and suppose B_i has type T_i:

1. If attribute B_i receives some constant b_i under α, then t_i is just the constant b_i.

2. If attribute B_i receives a key attribute K from S_1 under α, and either B_i is received by K under β, or B_i is involved in a join or selection condition in the body of some query in β, then t_i is just K_j, where K_j is the variable in the position of some key attribute K' in R that is guaranteed by Lemma 9 to receive attribute K and have the same value as B_i in every tuple in R.

3. Otherwise, t_i is just the constant $f(T_i)$.

Given any database instance e of $\kappa(S_2)$, the mapping β_κ is defined by $\beta_\kappa(e) = \pi_\kappa(\beta(\delta(e)))$. Clearly β_κ is a conjunctive query map since each (conjunctive) query in δ can be substituted for the appropriate relations in the body of each query in β.

The mapping β_κ must map a database instance in the range of α_κ back to its preimage under α_κ (recoverying the original database instance). Recall that α_κ creates values for the nonkey attributes that are projected out of some instance of $\kappa(S_1)$, and applies the map α to the resulting instance of S_1. This results in an instance of S_2 for which the nonkey attributes are then projected out to form the result of the map α_κ.

The map δ above recreates these nonkey attribute values that were projected out, producing an instance of S_2. While there typically is not sufficient information in the key attributes to recreate the missing nonkey attribute values precisely, the following lemma shows that δ recreates the values accurately for any nonkey attributes that can affect the result of applying the map β to the resulting instance with the values recreated.

Lemma 10. *Let S_1 and S_2 be keyed schemas. If e is a database instance of S_2 such that there is some database instance d_κ of $\kappa(S_1)$ satisfying $e = \alpha(\gamma(d_\kappa))$, then $\beta(\delta(\pi_\kappa(e)) = \beta(e)$.*

Proof. First note that $\pi_\kappa(\delta(\pi_\kappa(e))) = \pi_\kappa(e)$, because δ creates values for the nonkey attributes of e that are missing from $\pi_\kappa(e)$, and the outer application of π_κ just projects them back out again. Thus, $\delta(\pi_\kappa(e))$ and e have the same number of tuples in each relation, with identical key values. We now claim that if there are tuples τ in e and τ' in $\delta(\pi_\kappa(e))$ in the corresponding instances of the same relation that agree on key values, but disagree on nonkey attributes, then those tuples disagree on nonkey attributes that are neither received by any attributes in S_1 under β, nor participate in any selection or join conditions in any queries in β.

Suppose to the contrary that a pair of tuples, τ in e and τ' in $\delta(\pi_\kappa(e))$, exist in the corresponding instances of the same relation in each database instance and agree on their key values, but disagree on some arbitrary nonkey attribute B that either is received by some attribute in S_1 under β, or participates in a join or selection condition in a query in β. Let B have type T. By hypothesis there is some d_κ such that $e = \alpha(\gamma(d_\kappa))$. If attribute B receives some constant b under α, then attribute B of τ has value b from the query mapping and attribute B of τ' has value b by construction of δ, a contradiction. If attribute B receives a nonkey attribute under α, then attribute B of τ has value $f(T)$ by construction of γ and attribute B of τ' has value $f(T)$ by construction of δ, again a contradiction. If attribute B receives a key attribute K under α, then by Lemma 9 there is a key attribute K' in R such that K' has the same value as attribute B in the tuple τ. By the construction of δ, attribute B also has the same value as K' in τ', again a contradiction, establishing the claim.

We have shown that $\delta(\pi_\kappa(e))$ and e have the same number of tuples in each relation, with identical key values, and either the two instances in fact are equal, or corresponding tuples that agree on key values may disagree only on nonkey attributes that are not received by any attribute in S_1 under β and that do not participate in any join or selection conditions in any queries in β. But then, $\beta(\delta(\pi_\kappa(e))) = \beta(e)$, and the lemma follows. ∎

With the previous lemma established, we are equipped to prove the theorem of preservation of dominance by reduction to keys.

THEOREM 11. *If S_1 and S_2 are keyed schemas and $S_1 \preccurlyeq S_2$, then $\kappa(S_1) \preccurlyeq \kappa(S_2)$.*

Proof. Let $S_1 \preccurlyeq S_2$ by (α, β). We show that $\kappa(S_1) \preccurlyeq \kappa(S_2)$ by $(\alpha_\kappa, \beta_\kappa)$. It suffices to show that $\beta_\kappa \circ \alpha_\kappa$ is the identity map on $i(\kappa(S_1))$. Suppose not. Then there is some database instance d_κ of $\kappa(S_1)$ such that $\beta_\kappa(\alpha_\kappa(d_\kappa)) \neq d_\kappa$. Noting that $d_\kappa = \pi_\kappa(\gamma(d_\kappa))$ by the definition of γ, and substituting the definitions of β_κ and α_κ, we have that $\pi_\kappa(\beta(\delta(\pi_\kappa(\alpha(\gamma(d_\kappa)))))) \neq \pi_\kappa(\gamma(d_\kappa))$. Since two database instances that disagree on their keys cannot be the same instance, we infer that $\beta(\delta(\pi_\kappa(\alpha(\gamma(d_\kappa))))) \neq \gamma(d_\kappa)$. Now, if we let $d = \gamma(d_\kappa)$, we have $\beta(\delta(\pi_\kappa(\alpha(d)))) \neq d$. But letting $e = \alpha(d)$, Lemma 10 ensures that $\beta(\delta(\pi_\kappa(e))) = \beta(e)$, that is $\beta(\delta(\pi_\kappa(\alpha(d)))) = \beta(\alpha(d))$, whence $\beta(\alpha(d)) \neq d$. This contradicts that $\beta \circ \alpha$ is the identity map on $i(S_1)$, and the theorem follows. ∎

3.4. *Preservation of Functional Dependencies*

In this section we show that if $S_1 \preccurlyeq S_2$, then it is not possible for a functional dependency to fail on S_1 while simultaneously holding among the corresponding attributes in S_2, where the attributes of the dependency are encoded. The proof is based on the idea that if a functional dependency fails on S_1, then it fails on a nonempty attribute-specific database instance of S_1. Such an instance must be encoded in a nonempty instance in S_2, and because the instance of S_1 is attribute-specific, we can infer that the query mapping that encodes the instance of S_1 in an instance of S_2 must have a very restricted form from which the result is established in a straightforward manner.

THEOREM 12. *Let S_1 and S_2 be keyed schemas such that $S_1 \preccurlyeq S_2$ by (α, β) for conjunctive query mappings α and β. Suppose that $Y \to B$ holds for some relation R in schema S_2 for attribute B and attribute set Y in all instances in the range of α. Further suppose B is received by some attribute A under β and every attribute in Y is received under β by an attribute in some set X of attributes in S_1. Then it follows that the functional dependency $X \to A$ must hold in schema S_1.*

Proof. Suppose not. Then there exist keyed schemas S_1 and S_2 such that $S_1 \preccurlyeq S_2$ by (α, β) for conjunctive query mappings α and β, and a relation R in S_2 with Y a superkey of R and B an attribute of R. Further, there is a set X of attributes such that every attribute in Y is received by some attribute in X under β and an attribute A that receives B, but the dependency $X \rightarrow A$ fails on S_1. These relationships are shown in the Fig. 2.

By Lemma 5, every attribute in the set Y receives some attribute in X under α and B receives attribute A under α. Let d be an attribute-specific database instance of schema S_1 such that none of the values in the database d appear as constants in the queries in α or β, all of the relation instances in d are nonempty, and the dependency $X \rightarrow A$ fails in d. Lemma 3 guarantees that such a d always exists. Let q be the query in α that defines the instance of relation R under α. First, we claim that the instance of R in $\alpha(d)$ must be nonempty. If not, then the relations in S_1 containing either attribute A or any attribute in X would be empty in $\beta(\alpha(d)) = d$, since each attribute in Y is received by an attribute in X under β, and B is received by A under β. This is a contradiction, establishing the claim.

Since the dependency $X \rightarrow A$ fails, and all the attributes in X are received by an attribute in Y, and B receives A, there must be some join or selection conditions that filter out the necessary tuples so that $Y \rightarrow B$ holds in R. If there is a selection of an attribute as a constant value, this selection will fail because no query constants are in the database instance d, whence R will be empty in $\alpha(d)$, a contradiction. There also can be no column selection (that is, a selection that selects tuples that agree on two columns of the same relation) since R would again be empty in $\alpha(d)$ on account of d being an attribute-specific instance. Similarly, any join condition that is not an identity join condition will fail because d is an attribute-specific instance, whence R would be empty in $\alpha(d)$, again a contradiction.

Thus, q has no selection conditions, and the only join conditions are identity join conditions. By Lemma 2, there is a product query $\hat{q} \sqsubseteq q$ such that all of the relations that occur in the body of q also occur in the body of $\hat{q}$, and the dependency $Y \rightarrow B$ holds in the relation resulting from evaluating $\hat{q}$ over database d. Since, $\hat{q}$ can only contain cross-product and projection operations, this is a contradiction,

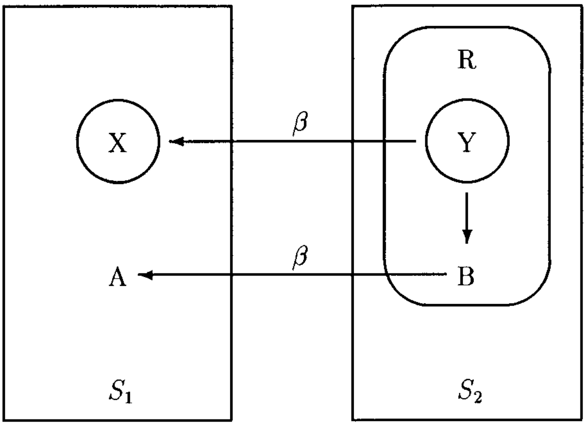

FIG. 2. Functional dependency relationships.

since a functional dependency that does not hold over any relation in a set of relations (in this case, the relations occurring in the body of $\hat{q}$) cannot hold in their cross product, and the theorem is established. ▌

3.5. *Conjunctive Query Schema Equivalence Theorem*

We now are ready to prove the central result of the paper, namely that keyed schemas are equivalent by conjunctive query maps if and only if they are isomorphic.

THEOREM 13. *If S_1 and S_2 are keyed schemas, then $S_1 \equiv S_2$ if and only if S_1 and S_2 are isomorphic.*

Proof. One direction is trivial; that is, if two schemas are isomorphic, then clearly they are equivalent. For the other direction, let S_1 and S_2 be keyed schemas with $S_1 \equiv S_2$. Then $S_1 \preccurlyeq S_2$ and $S_2 \preccurlyeq S_1$, and by Theorem 11, $\kappa(S_1) \preccurlyeq \kappa(S_2)$ and $\kappa(S_2) \preccurlyeq \kappa(S_1)$. Thus, $\kappa(S_1) \equiv \kappa(S_2)$. Note that all of the attributes of an unkeyed schema (schema with no depenencies) always implicitly form a key, so that $\kappa(S_1)$ and $\kappa(S_2)$ can be viewed as unkeyed schemas. The fact that the set of all the attributes of any relation in either schema forms a key is implicit. Thus, from the result of Hull [9] mentioned above, that states that equivalent unkeyed schemas are isomorphic, we conclude that $\kappa(S_1)$ and $\kappa(S_2)$ must be isomorphic.

But this means that S_1 has the same number of relations as S_2, and, for every relation R_1 in S_1, if R_1 has key K, where K is a set of attributes, then there is a corresponding relation R_2 in S_2 with the same key, up to renaming and re-ordering of attributes. The same is true for S_2: corresponding to every relation in S_2 there is a corresponding relation in S_1 with the same key (up to renaming and re-ordering of attributes).

It remains to be shown that the two schemas also agree on nonkey attributes. First, we claim that the number of occurrences of any attribute type among the nonkey attributes in one of the schemas must be the same as the number of occurrences of the same attribute type among nonkey attributes in the other schema. Without loss of generality, suppose to the contrary, that there is some attribute type A in S_1 that occurs a greater number of times among non-key attributes in S_1 than in S_2. Let (α, β) be the pair of conjunctive query maps establishing $S_1 \preccurlyeq S_2$.

By Lemma 4, each nonkey attribute of type A in S_1 must be received by some attribute in S_2 under α. Because there are more nonkey attributes of type A in S_1 than in S_2, it must be that there is some (key or nonkey) attribute A_0 with type A in S_2 that receives more than one attribute with type A in S_1, under the map α. Call the relation in which A_0 occurs R_0, and let $A_1, ..., A_n$ be the attributes of type A in S_1 that are received by A_0 under α. Note that $n \geqslant 2$, and $A_1, ..., A_n$ must all participate in the same join in the conjunctive query in α that defines R_0.

Let d be an attribute-specific instance of S_1 that includes some value $a \in \pi_{A_1}(d)$ that is not among the constants used in any of the queries in α or β. Such a d is guaranteed to exist by Lemma 3. Then the join in which the A_i, $i = 1, 2, ...$, participate must include a join condition that fails for the constant a, since the database state is attribute-specific, whence a is not found in the instance $\alpha(d)$. However, β must map the instance $\alpha(d)$ back to d, since $\beta \circ \alpha$ is the identity map on $i(S_1)$. But this implies that A_1 must receive some attribute with type A in S_2 under β, and a must be a value of that attribute in the instance $\alpha(d)$. This is a contradiction, which establishes the claim.

We have established that the two schemas have the same number of relations, with corresponding key sets, and the same number of occurrences of each nonkey attribute type. It remains only to be shown that the nonkey attributes cannot be rearranged so that the S_1 and S_2 are not isomorphic.

Let (α, β) be the pair of conjunctive-query maps establishing $S_1 \preccurlyeq S_2$. Let $\{R_1, R_2, ..., R_n\}$ be the set of all relations in schema S_2. For each i, let K_i be the set of attributes comprising the key of R_i, and let N_i be the remaining attributes of R_i. The functional dependency, $K_i \rightarrow N_i$, which is just the key dependency for R_i, holds for each i.

Now, for each i, let $\hat{K}_i$ be the set of all attributes that receive some attribute in K_i under β, and let $\hat{N}_i$ be the set of all attributes that receive some attribute in N_i under β. The sets $\hat{K}_i$, $\hat{N}_i$ must be pairwise disjoint, since otherwise there would an attribute in S_1 that receives two different attributes from S_2 under β, contradicting Lemma 6.

By Lemma 7 every attribute of each R_i is received by some attribute in S_1 under β. Thus, by Theorem 12, the functional dependencies $\hat{K}_i \rightarrow \hat{N}_i$ must hold in schema S_1 for each i. Since the only nontrivial functional dependencies holding on S_1 result from key dependencies, it follows that each $\hat{K}_i$ is a superkey of some relation, P_i in S_1, and that there are n such relations P_i.

For each i, and for every attribute type T, $\hat{N}_i$ must contain the same number of attributes of type T as N_i. If not, for some i there either must be two attributes in N_i that are received by the same attribute in $\hat{N}_i$ under β, contradicting Lemma 8, or there must be an attribute in N_i that is received by two different attributes in $\hat{N}_i$, contradicting Lemma 6. The same argument can be used to show that $\hat{K}_i$ has the same number of occurrences of any attribute type among its attributes as K_i, for each i.

Thus, every nonkey attribute in S_1 is an element of one of the $\hat{N}_i$, since for every attribute type T, there are the same number of nonkey attributes of type T in both schemas. This also means that each $\hat{K}_i$ is in fact the key of P_i, since if some $\hat{K}_i$ were a proper superkey, there would be additional nonkey attributes included in $\hat{K}_i$. Finally, because the only nontrivial functional dependencies holding on S_1 result from key dependencies, it follows that for each i, all of the attributes in $\hat{N}_i$ occur as nonkey attributes in P_i, and the theorem follows. ▮

4. CONCLUSIONS

Schema equivalence is a fundamental property of relational database schemas and is critical to the understanding of such problems as database design, data model translation, and multidatabase schema integration. However, while the notion of schema equivalence has been known for many years, there are surprisingly few results known that characterize the equivalence of relational schemas.

For example, a thorough understanding of database design or multidatabase schema integration would require, at a minimum, characterizations of schema equivalence for various classes of dependencies, such as primary keys, or primary keys plus referential integrity constraints, as well as other families of dependencies of interest. However, these problems remain open.

In this work, we have provided a number of results concerned with conjunctive query equivalence of relational schemas with primary keys, including having shown that two relational schemas with primary keys are equivalent by conjunctive query mappings if and only if they are isomorphic.

These results make substantial progress toward a characterization of the equivalence of schemas with primary keys (where the full relational algebra is available for schema mappings) and include the development of techniques that may be applicable to the solution of other problems concerning schema equivalence.

ACKNOWLEDGMENTS

The authors thank Rick Hull, the PODS-97 program committee members, and the anonymous referees for their insightful commentaries on earlier versions of this paper.

REFERENCES

1. J. Albert, Y. Ioannidis, and R. Ramakrishnan, Conjunctive query equivalence of keyed relational schemas (extended abstract), *in* "Proc. of the ACM Conf. on Principles of Database Systems, Tucson, AZ, 1997."
2. A. K. Arora and C. R. Carlson, The information preserving properties of relational database transformations, *in* "Proc. of the Int'l VLDB Conf., Rio de Janeiro, 1979."
3. P. Atzeni, G. Aussiello, C. Batini, and M. Moscarini, Inclusion and equivalence between relational database schemes, *Theoret. Comput. Sci.* **19** (1982), 267–285.
4. C. Batini, M. Lenzerini, and S. B. Navathe, A comparative analysis of methodologies for database schema integration, *ACM Computing Surveys* **18** (1986), 323–364.
5. C. Beeri, P. A. Bernstein, and N. Goodman, A sophisticate's introduction to database normalization theory, *in* "Proc. of the Int'l VLDB Conf., West Berlin, 1978."
6. C. Beeri, A. O. Mendelzon, Y. Sagiv, and J. D. Ullman, Equivalence of relational database schemes, *SIAM J. Comput.* **10** (1981), 352–370.
7. E. F. Codd, A relational model of data for large shared data banks, *Comm. ACM* **13** (1970), 377–387.
8. E. F. Codd, Further normalization of the data base relational model, *in* "Data Base Systems" (R. Rustin, Ed.), Prentice–Hall, Englewood Cliffs, NJ, 1972.
9. R. Hull, Relative information capacity of simple relational database schemata, *SIAM J. Comput.* **15** (1986), 846–886.
10. T.-W. Ling, F. W. Tompa, and T. Kameda, An improved third normal form for relational databases, *ACM Trans. Database Systems* **6** (1981), 329–346.
11. J. W. Lloyd, An introduction to deductive database systems, *Austral. Comput. J.* **15** (1983), 52–57.
12. D. Maier, "The Theory of Relational Databases," Comput. Sci. Press, Rockville, MD, 1983.
13. D. Maier, A. O. Mendelzon, F. Sadri, and J. D. Ullman, Adequacy of decompositions of relational databases, *J. Comput. System Sci.* **21** (1980), 368–379.
14. R. J. Miller, Y. Ioannidis, and R. Ramakrishnan, The use of information capacity in schema integration and translation, *in* "Proc. of the Int'l VLDB Conf., Dublin, 1993."
15. J. Rissanen, On equivalences of database schemes, *in* "Proc of the ACM Conf. on Principles of Database Systems, Los Angeles, CA, 1982."

16. A. Rosenthal and D. S. Reiner, Theoretically sound transformations for practical database design, *in* "Proc. of the Int'l Conf. on the Entity-Relationship Approach, New York, 1987."

17. A. Rosenthal and D. S. Reiner, Tools and transformations—rigorous and otherwise—for practical database design, *ACM Trans. Database Systems* **19** (1994), 167–211.

18. A. P. Sheth and J. A. Larson, Federated database systems for managing distributed, heterogeneous, and autonomous databases, *ACM Computing Surveys* **22** (1990), 183–236.

19. C. Zaniolo, A new normal form for the design of relational database schemata, *ACM Trans. Database Systems* **7** (1982), 489–499.