Relational Database Systems

Catherine M. Ricardo
Iona College

GLOSSARY

attribute A characteristic of an entity, represented in the relational model as a column of a table.
candidate key An attribute or minimal combination of attributes that could be used to uniquely identify tuples in a relation.
domain A set of possible values for an attribute.
entity A person, place, event, concept, or other "thing" which is to be represented by data in the database.
foreign key An attribute or set of attributes within a relation that matches the primary key of some, usually different, relation.
normalization A process for designing relational schemas with minimal redundancy and data anomalies.
primary key An attribute or combination of attributes that is used to uniquely identify tuples in a relation.
relation A set of *n*-tuples, each of which represents facts about an entity or relationship.
relationship An association or logical connection between entities.
schema The structure of a relational database, specified by writing the name of each relation and a parenthesized list of its attributes, with primary keys and foreign keys indicated. The schema also includes domains, views, character sets, constraints, stored procedures, authorizations, and other related information.
superkey Any attribute or set of attributes that uniquely identifies an entity.
table A physical representation of a relation, consisting of rows that correspond to entities and columns that correspond to attributes.
tuple A row of a table, corresponding to a single record.

I. INTRODUCTION

The relational model is currently the most widely used model for database management systems. The model itself is powerful but conceptually simple, and it is based on the mathematical notion of a relation. In the theoretical model, data are represented by relations, which are sets of tuples of attributes and their values. The relations are physically implemented as tables. The columns of the tables stand for data attributes. Each row of the table represents a tuple or record, which is a collection of related data values for the attributes. When discussing the theoretical model, we use the terms relation, attribute, and tuple. However, in the implementation of the model, these constructs are physically represented as tables, columns, and rows, respectively. Relationships between tables are represented by the use of foreign keys. The structure of a relational database is described in its schema, often presented as a list of its tables, each having a list of its attributes, with the primary keys and foreign keys indicated. The schema also includes constraints, domains, views, character sets, authorizations, stored procedures, and other related information. A design

process called normalization is used to help produce a good relational schema. A number of data languages can be used with the relational model, but SQL, Structured Query Language, is the standard relational language. There are many relational database management systems available; among them the best-selling database management products are Access, Oracle, DB2, SQL Server, and Sybase.

II. HISTORY OF THE RELATIONAL MODEL

The relational model was described by E.F. Codd in his 1970 paper, "A Relational Model of Data for Large Shared Data Banks." Existing database management systems at that time were hierarchical and network model systems. These systems were outgrowths of earlier file management systems and were complex and difficult for users. Codd's seminal paper proposed a revolutionary change in the structure of databases, and it sparked a flurry of research projects to test his theories. An early relational model project, System R, was developed at the IBM research laboratory in San Jose, California, in the 1970s. A more theoretical early project was the Peterlee Relational Test Vehicle, developed at the IBM UK Scientific Laboratory. Another early implementation of the model was INGRES, developed at the University of California at Berkeley. All of these projects resulted in many research papers and projects dealing with issues in the implementation of the relational model, including data representation, data languages, transaction management, user interfaces, and others. In the early 1980s microcomputer-based implementations of database management systems, overwhelmingly based on the relational model, became available. Since that time, the relational model has gained wide acceptance and has continued to be the subject of research studies to extend its capabilities. Extensions include efforts to add object-oriented concepts that can support more complex data, to capture more semantic information about the data, and to add intelligence to database systems.

III. THEORETICAL FOUNDATIONS OF THE RELATIONAL MODEL

The relational model is based on some fundamental ideas from mathematics that Codd extended to the realm of database systems.

A. Sets and Domains

A domain is a finite or infinite set from which values can be chosen. For example, the set of all integers is an infinite domain. The set of names, {Ann, Bob}, is a finite domain. The set of all integers between 5 and 7 inclusive, {5,6,7}, is another finite domain.

B. Cartesian Product and Tuples

For two domains D1 and D2, the Cartesian product, denoted D1 × D2, is defined as the set of all ordered pairs such that the first element is a member of D1 and the second element is a member of D2. That is,

$$D1 \times D2 = \{(d1,d2) | d1 \in D1, d2 \in D2\}.$$

If we let D1 be the set {Ann, Bob} and D2 be the set {5,6,7} then the Cartesian product is the set of ordered pairs that can be formed by choosing the first from D1 and the second from D2,

$$D1 \times D2 = \{(Ann,5), (Ann,6), (Ann,7), (Bob,5), (Bob,6), (Bob, 7)\}.$$

If we have three domains, D1, D2, and D3, we define the Cartesian product as the set of ordered triples with the first element from D1, the second from D2, and the third from D3. For example, if we let D3 = {red,blue} and let D1 and D2 be the sets defined earlier, then

$$D1 \times D2 \times D3 = \{(Ann,5,red), (Ann,6,red), (Ann,7,red), (Bob,5,red), (Bob,6,red), (Bob,7,red), (Ann,5,blue), (Ann,6,blue), (Ann,7,blue), (Bob,5,blue), (Bob,6,blue), (Bob,7,blue)\}.$$

If we have n domains, D1, D2, . . . , Dn, the Cartesian product is the set of ordered n-tuples, defined as

$$D1 \times D2 \times \ldots \times Dn = \{(d1, d2, \ldots, dn) | d1 \in D1, d2 \in D2, \ldots, dn \in Dn\}.$$

In the case of n domains we called the members of the Cartesian product n-tuples or just tuples, rather than ordered pairs or ordered triples.

C. Relations

If we have two domains D1 and D2, a relation can be defined as any subset of the Cartesian product of the domains. In our earlier example D1 × D2 had 6 ordered pairs. We can choose any number of these to form a relation, R. For example, we might choose

$$R = \{(Ann,5), (Bob,5)\}.$$

Sometimes it is possible to define the relation by giving a rule for the selection of members. For example, the following is an equivalent definition of R:

$$R = \{(d1,d2) \mid d1 \in D1, d2 \in D2, \text{ and } d2 = 5\}.$$

A second relation, S, could be defined simply by listing the members, as

$$S = \{(Ann,6), (Bob,6), (Ann,7), (Bob,5)\}.$$

For $D1 \times D2 \times D3$, shown earlier, an example of a relation might be

$$T = \{(Ann,5,red), (Bob,6,blue)\}.$$

For n domains, a relation would be any number of the n-tuples that we choose to be members of the relation.

Codd made a connection between relations in mathematics and stored facts in a database. For example, suppose D1 = {Ann, Bob} represents the set of tenants in an apartment building. Suppose D2 = {5,6,7} represents the apartment numbers. Suppose {red, blue} represents the set of favorite colors that people have. Then we could choose the tuples of the relation to correspond to facts about the occupants of the apartments and their favorite colors. For example, the relation

$$T = \{(Ann,5,red), (Bob,6,blue)\}$$

would mean that Ann occupies apartment 5 and her favorite color is red, while Bob occupies apartment 6 and his favorite color is blue. Obviously, when we create relations in a database we are not choosing tuples from the Cartesian product arbitrarily. Instead, we choose the tuples that correspond to values from the domains that represent facts. The name "relational" refers to relations of this type. They are the sets of tuples that represent facts that we wish to store in the database.

IV. STRUCTURE OF THE MODEL

A. Attributes

In the relational model, the word attribute has essentially the same meaning as it does in other models, such as the entity-relationship model. Attributes are data items of interest. Examples of attributes are customer names, balances of accounts, dates of transactions, and so on.

B. Tables

Relations are physically represented by means of two-dimensional tables in which the columns represent the attributes and the rows are the tuples of the relation. Each table represents the data for an entity set or for a relationship set. For example, Figure 1 shows a database with three tables. The Customer table displays the facts about customers. Each row lists the values of the attributes for a single customer. The column names are chosen to describe the attributes they represent. For example, the first row tells us that the customer with ID of C101 is named Martinez, lives in New York, and has a credit rating of 20. Similarly, the Item table gives facts about items in inventory. The Order table describes orders, including the customers who placed them and the items ordered. We assume each order is for only one item.

1. Domains

Domains are the sets from which values of attributes are chosen. However, in the relational model a domain must be an elementary data type, consisting of atomic, indivisible units. Complex types that can be broken down into simpler components are not permitted. Domains can be either finite or infinite. For example, the domain for creditRating might be specified as the (finite) set of positive integers between 1 and 20, while the domain of custName would be infinite, consisting of any string that could be interpreted as a name. A fundamental assumption is that all relations obey domain constraints, which means that the values in a column are taken from the domain of that attribute.

2. Null Values

For some attributes, the value for a particular entity may be unknown, missing, or not applicable. In that case the table is permitted to have a null value for that attribute. For example, the Item table in Figure 1 has no value for qtyOnHand for item I1004. This may mean that the item has never been ordered, or that we do not know how many we have, or that someone has neglected to enter the value. In the Customer table, if we had a new customer for whom we have not yet calculated a credit rating, we might enter the Customer record without a value for that column.

3. Keys

The notion of key is fundamental to relations. Just as in mathematical sets we must be able to distinguish

Customer	custID	custName	custCity	creditRating
	C101	Martinez	New York	20
	C105	Jones	London	20
	C110	LeBlanc	Paris	15
	C118	Wright	New York	10
	C125	LeBlanc	Montreal	18

Item	itemNo	itemName	price	supplier	qtyOnHand
	I1001	widget	2.99	Ace	200
	I1004	manifold	5.50	Acme	
	I1010	widget	3.75	Wright	150
	I1015	brace	6.80	Ace	16

Order	orderNo	custID	itemNo	qtyOrdered
	O10101	C101	I1004	50
	O10102	C105	I1010	30
	O10103	C118	I1015	5
	O10104	C101	I1001	30
	O10105	C125	I1015	10

Figure 1 The OrderSystem Database.

one element from another, with no repeats, it must always be possible to tell tuples apart by examining their values. Traditionally, a key consists of an attribute that has unique values. In a given relation there may be several such attributes, called candidate keys, and we can choose the one we want to use for the primary key. For example, if we are storing data about employees, and the company assigns a unique empID for each one, but we also store some national identifier such as Social Security Number, both of these attributes are candidate keys. We choose one of them as the primary key. If there is no single attribute with unique values, then a composite key, consisting of two or more attributes, may be used. Again, there may be several candidate composite keys. When we choose a composite key, it is essential that it be irreducible, i.e., that all the attributes be needed for uniqueness.

We sometimes use the primary key of one relation as an attribute of another relation, in order to show relationships between them. For example, in Figure 1, custID, which is the primary key of the Customer table, appears as an attribute of the Order table as well. The purpose of this repetition is to connect an order record to the customer who placed it. The custID is called a foreign key in the Order table. In general, a foreign key is an attribute that is the primary key of another relation, which we can call its home relation. In the Order table, itemNo is also a foreign key that refers to the Item table.

C. Schema

A schema for a relation consists of the name of the relation and its attributes. The relation schema is represented by a simple listing, such as

Customer(custID, custName, custCity, creditRating).

A database schema includes the schemas for all its relations. It is customary to underline the primary key for each relation. Often foreign keys are indicated in some way, such as by italics. We could give the database schema for the OrderSystem database shown in Figure 1 as

Customer(<u>custID</u>, custName, custCity, creditRating)

Item(<u>itemNo</u>, itemName, price, supplier, qtyOnHand)

Order(<u>orderNo</u>, *custID, itemNo,* qtyOrdered).

The schema is a relatively unchanging aspect of a database. It is the underlying structure, modified only rarely, when new data requirements arise. The actual data in the database, the records of actual customers, items, and orders, containing values such as those

shown in the bodies of the tables in Figure 1, are called an instance of the database. The database instance changes constantly, as new orders are placed, new customers or new items are added, old records are deleted, or data item values are changed. Some authors use the word "intension" to refer to the schema or stable structure, and "extension" to refer to the instance. Strictly speaking, the schema also includes domains, views, character sets, constraints, stored procedures, authorizations, and other related information.

D. Characteristics of Relations

Relations have certain properties because of the way they are defined to correspond to mathematical relations. Since the underlying domains are scalar data types, each attribute must have a single irreducible value for each tuple. When we see the table in spreadsheet-type view, each cell of the table must have just one value. For example, in the Item table in Figure 1, we can list only one supplier for each item. The order of tuples is immaterial. Although it may be convenient to list tuples in order by primary key, the relation remains the same if we display the tuples in a different order, because the elements of a set are unordered. The order of columns is also immaterial. Note that this assumption is a departure from the usual mathematical definition of relation, in which the individual elements within the tuples in the Cartesian product must be in order. In the relational data model, it is the name of the attribute, rather than its position, that indicates its meaning. Although it is usual to display a table's columns in the order shown in its schema, the relation remains the same if we switch the positions of columns in the schema. Of course when we display the data, every row in the table will have the attribute values placed in a consistent fashion to correspond to the position of the column names. There are no duplicate tuples, just as there are no repeating elements in a set. This property guarantees that it is always possible to distinguish between tuples, which means there will always be a key, although it may be a composite one.

E. Integrity Rules

Two fundamental rules defined by Codd are entity integrity and referential integrity. Entity integrity means that all components of the primary key must always have an actual value for each tuple of the table. No null values are permitted for any part of the primary key. This rule guarantees that we can always tell tuples apart. The second rule is referential integrity, which refers to foreign keys. Any nonnull value of a foreign key must match a value in the home relation. For example, a custID value in the Order table of Figure 1 must be one of the values of custID in the Customer table. An order record cannot refer to a customer who does not exist. In some cases, the foreign key value is permitted to be null.

F. Views

The actual tables in a relational database are called base tables. A database administrator can grant users permission to access and/or modify the contents of those tables. In some cases, the administrator may wish to allow a user to access only a subset of a base table, or a virtual table created by combining parts of base tables, or by performing summary or other group functions on them. The database administrator can create a view, which can be thought of as either a window into a base table, or as a virtual table. Although the definition of the user's view is stored permanently, the view itself is created dynamically when the user requests access to it. The views that act as windows into a base table can change dynamically as the underlying table is updated. Others are "snapshots," and their contents remain static during the user's session, although they are updated for each new session. Users can perform queries on views. Depending on the nature of the view, it may be possible for the user to make updates that affect the underlying base tables. This might be the case when the user's view contains the primary key and all other required fields of a table, and the view is a window into a base table. A view is both a security mechanism, protecting hidden attributes, and a facility for the user, simplifying his model of the database. It also provides a way to insulate the user from changes to the database structure, a characteristic called logical data independence. For example, if the Customer table is restructured to include the customer's telephone number, a view may be defined that does not include the new attribute, allowing existing applications to continue functioning by using the view.

G. System Catalog

A relational database is self-describing. When the database designer executes the commands that create or

modify the structures of tables, views, and indexes, the system itself records a description of the database in its system catalog. The catalog is itself a small relational database that stores information about each relation, its attributes, its indexes, and any integrity constraints such as primary and foreign keys. One of the tables in the system catalog keeps track of the database's tables. Each row lists all the information about one of the tables in the database, including its name, number of attributes, the name of its creator, the date it was created, and so on. Another table describes all the attributes in the entire database. Each row lists an attribute, the table it belongs to, its data type, and so on. Similar tables are kept for views, indexes, and constraints. The catalog also stores statistics about the size of tables, the size and number of values of indexes, the range of values for some attributes, and other data that are used by the query subsystem to plan efficient execution of queries. Access control information and usage information about users are also stored in the catalog. Since the catalog is a relational database, it can be queried using SQL or other query language.

V. RELATIONAL DATA LANGUAGES

A number of languages can be used to create, manipulate, and administer a relational database. These include SQL, Query by Example, and two theoretical languages, relational algebra and relational calculus. Some of the query languages are procedural, which means they specify the sequence of operations to be performed to retrieve the desired data. Others are declarative (nonprocedural) and specify what data are desired, but now how to retrieve it.

A. Relational Algebra

Relational algebra is a formal, theoretical language that provides a basic set of operators that can be applied to tables and that return another table as a result. Codd originally proposed eight basic relational algebra operators, but the set has been extended. Relational algebra is a procedural language. Although it is not implemented in its native form in commercial database management systems, it is useful because it enables us to understand the basic retrieval operations that are required for any query language. It also serves as a benchmark for other relational languages, in that any relational language that can perform the same operations as relational algebra is said to be relationally complete. The original language proposal included four set operations, Union, Intersection, Difference, and Cartesian Product. These operators are defined to be similar to the corresponding set-theoretic operations in mathematics. However, the elements of the sets in this case are tuples, the rows of the tables, rather than individual values.

1. Union

The relational algebra union operation on tables R and S, denoted R ∪ S, is a binary operator that is applied to two tables that are union-compatible, which means that they have the same attributes, defined on the same domains, and in the same order, on both tables. For example, consider the union-compatible tables shown in Figure 2:

AutoInsCust(<u>custID</u>, lname, fname, agentID)

LifeInsCust(<u>custID</u>, lname, fname, agentID).

We can infer that the tuples in the first table show information about customers of an insurance agency who own automobile insurance policies and the tuples of the second about customers who own life insurance policies. We assume a customer can own either or both types of policies, but a customer always has the same agent. The union of the two tables consists of the tuples that are in either AutoInsCust or LifeInsCust or in both. The result is shown in Figure 3. It lists customers who own either type of policy, or both. Observe that duplicates have been eliminated, so that those who own both types are listed only once in the result. If the two original tables have essentially the same attributes, with the same domains, but different names, it is possible to rename the attributes so that they match and then the union can be performed.

2. Intersection

The intersection operation, written R ∩ S, is another binary operator that is applied to union-compatible

AutoInsCust	**custID**	**lname**	**fname**	**agentID**
	C123	Smith	John	A444
	C456	Jones	Mary	A555
	C789	Adams	Sue	A444
LifeInsCust	**custID**	**iname**	**fname**	**agentID**
	C123	Smith	John	A444
	C789	Adams	Sue	A444
	C246	Barclay	Tom	A777

Figure 2 Union-Compatible Tables.

custID	lname	fname	agentID
C123	Smith	John	A444
C456	Jones	Mary	A555
C789	Adams	Sue	A444
C246	Barclay	Tom	A777

Figure 3 AutoInsCust ∪ LifeInsCust.

tables. It consists of the tuples that are in both of the tables simultaneously. For the tables in Figure 2, the intersection is shown in Figure 4. It consists of customers who own both automobile insurance and life insurance policies with the agency.

3. Difference

The difference between two union-compatible tables, R − S, is the set of tuples that belong to the first table, R, but not to the second, S. The result of AutoIns Cust − LifeInsCust, shown in Figure 5, is the set of tuples of AutoInsCust that are not also in LifeInsCust. These are the customers who own automobile insurance policies, but not life insurance policies. Note that, unlike Union and Intersection, Difference is not commutative, meaning you do not get the same results if you interchange the two tables.

4. Cartesian Product

As described previously, the Cartesian product of two sets, A and B, consists of all the ordered pairs that can be constructed with the first element coming from the first set, A, and the second element coming from the second set, B. We can extend that notion to tables, R and S, which have distinct attributes. The product of the tables consists of all tuples that can be formed by concatenating every tuple of R with every tuple of S. For example, consider the Customer table and the Item table in Figure 1. The Cartesian product, Customer × Item, is shown in Figure 6. The number of columns in the result is the sum of the number of columns in each of the two original tables. For the example shown, there are 4 columns in Customer and 5 in Item, so there are 9 columns in the result. The number of rows is the product of the number of rows in the original tables. Since there are 5 rows in Customer and 4 in Item, the result table shows 20 rows. Note that in the case where the two original tables have one or more attribute names that are identical, we can use the name of the original table as a prefix before the attribute name in the result, so that each of the columns in the result can have a unique name. For example, if custName and itemName were both simply called name in their respective tables, we could call the first Customer.name and the second Item.name in the result.

Codd also described several relational operators created for relational databases, including Select, Project, Equijoin, and Divide.

custID	lname	fname	agentID
C123	Smith	John	A444
C789	Adams	Sue	A444

Figure 4 AutoInsCust ∩ LifeInsCust.

custID	lname	fname	agentID
C456	Jones	Mary	A555

Figure 5 AutoInsCust − LifeInsCust.

5. Select

The selection operator is unary, which means it is applied to one table at a time. The result is a new table that has the same structure as the original. The operation takes rows from the original table that satisfy a specified condition, called the selection condition, producing a horizontal subset of the table. Symbolically, we write

$$\sigma_{\text{<selection condition>}}\ (\text{table-name}).$$

The Greek letter σ (sigma) is the symbol for Select. The subscript condition, sometimes written as the Greek letter θ (theta) and called the θ-condition, is a predicate involving some condition on the table, normally using a comparison operator and one or more attributes. For example, if we want to choose the rows of the Customer table in Figure 1 where the creditRating is greater than 15, we would write

$$\sigma_{\text{creditRating} > 15}\ (\text{Customer}).$$

The result of this operation is shown in Figure 7. The selection condition may use any of the standard comparison operators {=, <, <=, >, >=, ≠} to compare an attribute with a constant value or with another attribute. The logical connectives AND, OR, and NOT can also be used to make a compound selection condition. For example, if we wrote

$$\sigma_{\text{creditRating <=10 AND custCity} \neq \text{'New York'}}\ (\text{Customer})$$

custID	custName	custCity	creditRating	itemNo	itemName	price	supplier	qtyOnHand
C101	Martinez	New York	20	I1001	widget	2.99	Ace	200
C101	Martinez	New York	20	I1004	manifold	5.50	Acme	
C101	Martinez	New York	20	I1010	widget	3.75	Wright	150
C101	Martinez	New York	20	I1015	brace	6.80	Ace	16
C105	Jones	London	20	I1001	widget	2.99	Ace	200
C105	Jones	London	20	I1004	manifold	5.50	Acme	
C105	Jones	London	20	I1010	widget	3.75	Wright	150
C105	Jones	London	20	I1015	brace	6.80	Ace	16
C110	LeBlanc	Paris	15	I1001	widget	2.99	Ace	200
C110	LeBlanc	Paris	15	I1004	manifold	5.50	Acme	
C110	LeBlanc	Paris	15	I1010	widget	3.75	Wright	150
C110	LeBlanc	Paris	15	I1015	brace	6.80	Ace	16
C118	Wright	New York	10	I1001	widget	2.99	Ace	200
C118	Wright	New York	10	I1004	manifold	5.50	Acme	
C118	Wright	New York	10	I1010	widget	3.75	Wright	150
C118	Wright	New York	10	I1015	brace	6.80	Ace	16
C125	LeBlanc	Montreal	18	I1001	widget	2.99	Ace	200
C125	LeBlanc	Montreal	18	I1004	manifold	5.50	Acme	
C125	LeBlanc	Montreal	18	I1010	widget	3.75	Wright	150
C125	LeBlanc	Montreal	18	I1015	brace	6.80	Ace	16

Figure 6 The Cartesian Product: Customer × Item.

the result would be an empty table, since no row of Customer satisfies this condition.

6. Project

Project is a unary operator that produces a vertical subset of a table. The result is a new table with only the columns specified in a list of attributes, called the projection list, of the original table. For those columns, it eliminates duplicate rows so that only the unique combinations of values are returned. The general form is

$$\Pi_{\text{projection list}}\ (\text{table-name}).$$

The projection list can be a single column, as in

$$\Pi_{\text{supplier}}\ (\text{Item}).$$

Using the Item table shown in Figure 1, the result of this projection is shown in Figure 8. Note that the duplicate value for Ace was eliminated. If we project over more than one attribute, the unique combinations of values appear. For example,

$$\Pi_{\text{custName, custCity}}\ (\text{Customer})$$

produces the table shown in Figure 9. Even though some names are repeated and some cities are repeated, no given combination of values is repeated.

custID	custName	custCity	creditRating
C101	Martinez	New York	20
C105	Jones	London	20
C125	LeBlanc	Montreal	18

Figure 7 Example of Selection: $\sigma_{\text{creditRating}> 15}\ (\text{Customer})$.

supplier
Ace
Acme
Wright

Figure 8 Example of Projection: $\Pi_{\text{supplier}}\ (\text{Item})$.

custName	custCity
Martinez	New York
Jones	London
LeBlanc	Paris
Wright	New York
LeBlanc	Montreal

Figure 9 Projection over Multiple Attributes: $\Pi_{\text{custName, custCity}}$ (Customer).

7. Equijoin

The equijoin is a binary operator on two tables that have a common attribute. It is formed by concatenating rows from the first table with rows from the second table that have the same value for the common attribute. If A is the attribute that appears in both tables, R and S, the equijoin could be expressed as

$$R \mid\times\mid_{R.A = S.A} S.$$

Equivalently, the equijoin could be expressed as the result of performing the Cartesian product, and then selecting the rows that have the same value for the common attribute. For R and S, the equijoin is the same as

$$\sigma_{R.A = S.A} (R \times S).$$

As an example, we can form the equijoin of Customer and Order, since both tables have custID, giving the result shown in Figure 10. We could form the entire Cartesian product of the two tables, and then choose only those rows where Customer.custID = Order.custID. Note that if the two tables have attributes with a common domain, even if the attributes have different names, the equijoin can still be performed.

8. Division

Division is a relatively complex operator that is defined on two tables whose schemas are related so that one (the divisor) is a subset of the other (the dividend). The schema for the result (the quotient) is the set of attributes that appear on the dividend table but not on the divisor table. The tuples are the values of these attributes that appear on the dividend table with all values that appears on the divisor table. The general notation is

$$R \div S.$$

For example, assume we have the two tables shown in Figure 11. We note that all of the attributes of Skill (skillName) are contained in the HasSkill table (skillName, empName). Here, Skill is the divisor and HasSkill is the dividend. The division operation HasSkill ÷ Skill produces a quotient, a table showing the names of employees who have all the skills listed on the Skill table, as shown in Figure 12. Several other join operators have been added to relational algebra since Codd's original work. They include the theta-join, the natural join, and the outerjoin.

9. Theta Join

A theta join is a binary operation defined as the result of taking the Cartesian product of the two tables and then applying selection, using a θ-condition, to the result. The general form of a theta join is

$$R \mid\times\mid_{\theta} S.$$

From the definition, this is equivalent to $\sigma_{\theta} (R \times S)$. For example, if we wanted to take a theta join of the Customer and Item tables from Figure 1, with θ being creditRating > qtyOnHand, we would have the result shown in Figure 13. This result is obtained by applying the selection operator θ to the Cartesian product of the two tables, which we found in Figure 6. An equijoin could be considered a specialized form of the theta-join, in which the condition is equality.

10. Natural Join

When an equijoin is performed, the resulting table will always have two columns with identical values. If

Customer.custID	custName	custCity	creditRating	orderNo	Order.custID	itemNo	qtyOrdered
C101	Martinez	New York	20	O10101	C101	I1004	50
C101	Martinez	New York	20	O10104	C101	I1001	30
C105	Jones	London	20	O10102	C105	I1010	30
C118	Wright	New York	10	O10103	C118	I1015	5
C125	LeBlanc	Montreal	18	O10105	C125	I1015	10

Figure 10 Equijoin: Customer $\mid\times\mid_{\text{Customer.custID=Order.custID}}$ Order.

HasSkill	skillName	empName
	programming	Smith
	programming	Jones
	programming	Adams
	systems analysis	Smith
	systems analysis	Adams
	database administration	Smith
	database administration	Jones
	database administration	Adams

Skill	skillName
	programming
	systems analysis
	database administration

Figure 11 Tables for Division.

we remove one of the duplicate columns by means of a projection, so that the common column appears only once in the result, we have a natural join. Because this is the most common type of join, we use the shorthand notation |×| for it, without a subscript, as in

$$R \mid\times\mid S.$$

For example, Figure 14 shows Order |×| Item.

11. Outerjoin

The outerjoin is an extension of the natural join. The natural join is formed and then those tuples from either of the original tables that are not represented in the result because they had no matching tuples are added, with null values for attributes from the other relation. For example, if we took the natural join Customer |×| Order we would find that there is no Order tuple with custID of C110, so that Customer tuple would not be included in the natural join. In the outerjoin, we include such tuples, inserting null values for the attributes from the second table. The result is shown in Figure 15. In this example, only the left-hand table, Customer, had an unmatched tuple. If the right-hand table, Order, had unmatched tuples, then we would have added the unmatched tuples from that table as well. It is possible to distinguish three different outerjoins: the left outerjoin, in which we add only unmatched tuples from the left-hand table, the right outerjoin, in which we add only unmatched tuples from the right-hand table, and the full outerjoin, in which we add unmatched tuples from both tables, if any exist.

empName
Smith
Adams

Figure 12 Division: HasSkill ÷ Skill.

12. Composition of Relational Algebra Operations and Renaming

As we have seen, each relational algebra operation results in another relation, a property called closure. The resulting relation could therefore have been used to perform a second relational algebra operation, giving a sequence of two operations. In fact, we can carry out a sequence of several operations, each of which is performed using the result of the previous one. For example, using Figure 1, suppose we wished to find the names of all Customers who have ordered item I1004. We can see from the Order table that only the first tuple involves I1004. Choosing rows that have a specified value is a SELECT operation, so we write

$$\sigma_{\text{itemNo}='\text{I1004}'}(\text{Order})$$

for that part of the query. Once we have found the tuple or tuples having the correct itemNo value, we do a natural join with the Customer table to find the corresponding customer records. We indicate that the join is to be performed with the previous result by writing

$$(\sigma_{\text{itemNo}='\text{I1004}'}(\text{Order})) \mid\times\mid \text{Customer}.$$

Finally, we find the custName by applying a projection to this result, so that the entire expression becomes

$$\Pi_{\text{custName}}((\sigma_{\text{itemNo}='\text{I1004}'}(\text{Order})) \mid\times\mid \text{Customer}).$$

It is permissible to assign names to the intermediate results, a process called renaming, by a simple statement such as

$$\text{Temp1} \leftarrow \sigma_{\text{itemNo}='\text{I1004}'}(\text{Order}).$$

custID	custName	custCity	creditRating	itemNo	itemName	price	supplier	qtyOnHand
C101	Martinez	New York	20	I1015	brace	6.80	Ace	16
C105	Jones	London	20	I1015	brace	6.80	Ace	16
C125	LeBlanc	Paris	18	I1015	brace	6.80	Ace	16

Figure 13 Theta Join: Customer $|\times|_{\text{creditRating} > \text{qtyOnHand}}$ Item.

Then we could write

$$\text{Temp2} \leftarrow \text{Temp1} \; |\times| \; \text{Customer}$$

and

$$\text{Result} \leftarrow \Pi_{\text{custName}} \; (\text{Temp2}).$$

We can rename tables, attributes, or both using a Rename operator, usually designated by the Greek letter rho, ρ. If R is a table having attributes A1, A2, . . . , An and we wish to rename the table to S and/or the attributes to B1, B2, . . . , Bn, we write

ρ_S (R) to rename the table to S and keep all the attribute names

or

$\rho_{(B1, B2, \ldots, Bn)}$ (R) to keep the table name R but change attribute names to Bi

or

$\rho_{S(B1, B2, \ldots, Bn)}$ (R) to change the table name to S and the attribute names to Bi.

For example, to change the name of the Customer table to Cust and the names of some of the attributes, we could write

$$\rho_{\text{Cust(custNumber, custName, custCity, custCredit)}} \; (\text{Customer}).$$

We can then refer to Cust and its attributes in subsequent relational algebra expressions, such as

$$\text{Cust} \; |\times|_{\text{custNumber=custID}} \; \text{Order}.$$

13. Other Extensions of Relational Algebra

Relational algebra has been extended in many ways by various researchers. Among the most important extensions are aggregate functions such as SUM, AVG, MAX, MIN, and COUNT. The standard operators have also been extended to include systematic treatment of null values.

14. Query Optimization

Although relational database management systems use a variety of query languages, when the queries are implemented the resulting operations can be expressed in relational algebra or some equivalent language. It is sometimes possible to express the same relational algebra query using different operations or different sequences of operations. Some of these equivalent expressions are more efficient than others. The DBMS typically has a query optimizer to handle the task of finding and evaluating the alternative ways a query could be executed, producing a query plan that dictates which operations will be executed and in what order. To do so, the optimizer uses some common algebraic laws to develop alternative relational algebra expressions. It can be proven, for example, that joins are commutative. If R and S are two tables, then

$$R \times S = S \times R, \quad \text{and} \quad R \; |\times| \; S = S \; |\times| \; R.$$

Joins are also associative. If R, S, and T are three tables, then

$$(R \times S) \times T = R \times (S \times T) \quad \text{and}$$
$$(R \; |\times| \; S) \; |\times| \; T = R \; |\times| \; (S \; |\times| \; T).$$

orderNo	custID	itemNo	qtyOrdered	itemName	price	supplier	qtyOnHand
O10101	C101	I1004	50	manifold	5.50	Acme	
O10102	C105	I1010	30	widget	3.75	Wright	150
O10103	C118	I1015	5	brace	6.80	Ace	16
O10104	C101	I1001	30	widget	2.99	Ace	200
O10105	C125	I1015	10	brace	6.80	Ace	16

Figure 14 Natural Join: Order |×| Item.

custID	custName	custCity	creditRating	orderNo	itemNo	qtyOnHand
C101	Martinez	New York	20	O10101	I1004	50
C101	Martinez	New York	20	O10104	I1001	30
C105	Jones	London	20	O10102	I1010	30
C110	LeBlanc	Paris	15	null	null	null
C118	Wright	New York	10	O10103	I1015	5
C125	LeBlanc	Montreal	18	O10105	I1015	10

Figure 15 Outerjoin: Customer OUTERJOIN Order.

Set union is also both commutative and associative, as is set intersection. There are also laws governing selection, projection, and other operations. The laws involving selection are particularly useful, because selection usually produces a result that is significantly smaller than the original table. Therefore, if a selection can be done early in the query plan, the rest of the query operations may be performed on tables that are much smaller than the initial ones, resulting in considerable improvement in efficiency. The most common heuristic, or "rule of thumb" used by optimizers is to do selection as early as possible. For example, if we wanted to find the item number of all items ordered by customer Martinez, we could start by doing a natural join of the Order and Customer tables over the common column, custID, then do a selection to find the rows where the custName is Martinez, then do a projection over the itemNo. The relational algebra translation, using renaming, would be

$$\text{Temp1} \leftarrow \text{Customer} \mid\times\mid \text{Order}$$

$$\text{Temp2} \leftarrow \sigma_{\text{custName}=\text{'Martinez'}} (\text{Temp1})$$

$$\text{Result} \leftarrow \pi_{\text{itemNo}} (\text{Temp2}).$$

For the data shown, Temp1 would contain 5 records, each having 7 attributes. Temp2 would result in 2 records, still having 7 attributes. The result would have 2 records with a single attribute each.

However, if the selection is done first, we would get

$$\text{Temp1} \leftarrow \sigma_{\text{custName}=\text{'Martinez'}} (\text{Customer})$$

$$\text{Temp 2} \leftarrow \text{Temp1} \mid\times\mid \text{Order}$$

$$\text{Result} \leftarrow \pi_{\text{itemNo}} (\text{Temp2}).$$

Using this scheme, Temp 1 would have only 1 record. Temp 2 would have 2 records, with 7 attributes each, and the result would be as before. By doing the selection early, we have reduced the size of the first intermediate table considerably. Relational database management systems use algebraic equivalences like these to transform queries into more efficient equivalent forms. They then choose among possible query plans based on physical factors such as the existence of indexes, the order of records in tables, the selectivity of conditions, the number of values for attributes, and the number of records for tables.

B. Relational Calculus

Relational calculus is another formal, theoretical language for expressing relational database queries. It is equivalent to relational algebra, which means that any query that can be expressed in one of the languages can also be written in the other. Therefore, relational calculus is relationally complete. It is a nonprocedural language, allowing us to express what data we want without specifying how the retrieval is to be done. There are two forms, tuple relational calculus and domain relational calculus. Both use first-order logic, a branch of mathematics.

1. Tuple Relational Calculus

Tuple relational calculus uses variables that range over the tuples of a relation. For example, the query, "Find the custID and custName of all customers with credit rating greater than 15" might be written in a much-simplified version of tuple relational calculus as

$$\{\text{C.custID, C.custName} \mid \text{Customer(C) and C.creditRating} > 15\}.$$

Here C is a tuple variable and we are specifying that we want the set of all custID and custName values for all the tuples (values of C) where C ranges over the Customer table and the creditRating value of C is greater than 15. As in relational algebra, predicates can include the comparison operators $<$, $<=$, $>$, $>=$, $\neq$ and the logical connectives AND, OR, and NOT. The expressions can also include the logical quantifiers

FORALL and THERE EXISTS. There are strict rules about the form of an expression in tuple relational calculus to guarantee that the expression will be safe, which means it will not lead to an infinite result.

2. Domain Relational Calculus

In domain relational calculus the variables range over the domains of attributes. The general form of a query is

$$\{<x1, x2, \ldots, xn> \mid P(x1, x2, \ldots, xn)\}.$$

This means the set of all domain variables x1, x2, . . . , xn for which the predicate P(x1, x2, . . . , xn) is true.

For example, to find complete Item records for which the qtyOnHand is greater than 150, we could write

$$\{<N, A, P, S, Q> \mid <N, A, P, S, Q> \in \text{Item and } Q > 150\}.$$

If we wanted only specific fields of these item records, we would have to add the condition that there exist values for the other attributes for which the predicate is true. For example, to find only the itemNo and itemName for items with qtyOnHand above 150, we would write

$$\{<N, A> \mid (\text{THERE EXISTS } P)\ (\text{THERE EXISTS } S)\ (\text{THERE EXISTS } Q)$$
$$<N, A, P, S, Q> \in \text{Item and } Q > 150\}.$$

C. SQL

SQL, Structured Query Language, is the standard data language for relational database management systems. It is based on tuple relational calculus, although it also has similarities to relational algebra. Most relational database management systems support some version of SQL. SQL has data definition, data manipulation, user authorization, and transaction control facilities. In the data manipulation language, the query statement, the SELECT, incorporates the same fundamental operations as relational algebra. Its syntax is quite similar to the relational algebra statements as well. For example, the query we considered earlier using relational algebra, Find the names of all customers who have ordered item I1004, can be written in SQL as

```
SELECT custName
FROM Customer, Item
WHERE itemNo = 'I1004' AND Customer.custID =
                    Item.custID;
```

D. Query by Example

Query by Example was originally developed by IBM to provide a user-friendly graphical interface for relational databases. It is based on domain relational calculus. Although IBM offers a commercial version that is very close to relational calculus, QBE-type interfaces are available with many other microcomputer-based database management systems, including Microsoft's Access. To design a query in Access, the user is presented with windows that can be modified. In one window the user can choose the tables to be included in the query. They will be displayed there along with their attributes and relationships. The second window is a form that the user can fill in by clicking on attribute names from the tables and adding criteria or conditions. There are options for sorting and other facilities. Once the query is designed, it can be executed immediately, and the results can be displayed or named and saved for use in another query or for reports. There is also an option to allow the user to view the SQL version of the query.

E. ODBC and JDBC

Standards exist for allowing application programs to access data in databases, using the SQL language. Microsoft's Open Database Connectivity (ODBC) technology provides standards for a common interface for accessing SQL-based databases. Using the interface, an application program can access different databases, having various database management systems and operating systems environments, using the same code. ODBC database drivers are provided by most database management system software. For applications written in Java, the de facto standard is JDBC.

VI. NORMALIZATION

Modeling is the process of designing a schema for a database. One method that is widely used in relational modeling is normalization, a technique for creating a database schema that minimizes redundancy and that is free of certain undesirable properties called anomalies. There are several normal forms, each with its own restrictions. The normalization process uses a series of tests in which each relation in a proposed schema is examined to determine whether it satisfies the requirements of each normal form. If it does not conform to the normal form, it may be possible to decompose the relation into two or more equivalent

relations that satisfy the requirements of the form. It is important to note that normalization is concerned with the schema, the intension of the database. An instance or extension may be used to provide examples to guide us to see problems with the schema. An instance may also provide a counterexample, showing that the relation is not in the form required, but an instance cannot prove that a relation is in the normal form. Such proof requires that we examine some fundamental aspects of the relations and attributes. These aspects include functional dependency, multivalued dependency, and properties of decompositions. Codd first proposed normalization and defined the first three normal forms, based on functional dependency. Along with Boyce, he refined the third normal form definition into Boyce–Codd normal form. Other researchers independently identified multivalued dependencies, which are involved in fourth normal form, and join dependencies, which are used in fifth normal form.

A. Functional Dependency

If A and B are attributes or sets of attributes of relation R, then B is said to be functionally dependent on A if each value of A in R has associated with it exactly one value of B. Equivalently, if two tuples of R have the same A value, they must have the same B value. We write this as A → B, read as A functionally determines B. The attribute or attribute set to the left of the arrow is called the determinant. In normalization, the first task is to determine all the functional dependencies, sometimes called FDs, among the data items. For example, consider the following set of data items for a company with many employees, several departments, and many projects:

{empID, SSN, empName, deptID, deptMgr, jobTitle, rating, projID, projName, projDir, budget, salary, hoursAssigned}.

We assume that SSN represents Social Security number or an equivalent universal identifier, and that empID is a unique identifier issued within the company. We also assume that each employee may work on many projects simultaneously, and that hoursAssigned records the number of hours an employee works on a project, and rating the worker's performance rating on that project. Each project has one director, one name, one budget, and many employees. Project names are unique. Names of persons are not unique. An instance of the Company table is shown in Figure 16.

Given those assumptions, we have the following FDs. Attributes written on the right of the arrow are individually determined by attributes or sets of attributes on the left.

empID → SSN, empName, deptID, deptMgr, jobTitle, salary

SSN → empID, empName, deptID, deptMgr, jobTitle, salary

projID → projName, projDir, budget

projName → projID, projDir, budget

deptID → deptMgr

{empID, projID} → rating, hoursAssigned, plus all attributes functionally dependent on either empID or projID individually

{SSN, projID} → rating, hoursAssigned, plus all attributes functionally dependent on either SSN or projID individually

{empID, projName} → rating, hoursAssigned, plus all attributes functionally dependent on either empID or projName individually

{SSN, projName} → rating, hoursAssigned, plus all attributes functionally dependent on either empID or projName individually.

We ignore "trivial" functional dependencies, in which the attribute(s) on the right is included in the determinants, such as

empID → empID or

{empName, salary} → empName.

Functional dependencies help us to identify candidate keys for a table. If we look at the list of FDs we can identify attributes or sets of attributes such that every attribute is functionally dependent on them. They therefore identify each tuple uniquely. Such a set is called a superkey. For our example, we have many superkeys, some of which are

{SSN, projID}

{empID, projID}

{SSN, projName}

{empID, projName}, but also

{empID, empName, projID}

{projID, SSN, deptID} and others.

Each of these superkeys uniquely identifies each row of the table. That is, if you were told the values of the

empID	SSN	empName	deptID	deptMgr	jobTitle	rating	projID	projName	projDir	budget	salary	hours Assigned
101	111223344	Adams	10	Munez	programmer	4	J10	Atlantic	Quinn	100000	45000	20
101	111223344	Adams	10	Munez	programmer	5	J25	Pacific	Ryan	200000	45000	10
102	234567890	Burns	12	Nolan	DBA	4	J10	Atlantic	Quinn	100000	60000	30
103	222233344	Cabot	11	Peyton	DBA	3	J30	Arctic	Smith	200000	60000	10
104	445566778	Adams	12	Nolan	programmer	3	J25	Pacific	Ryan	200000	50000	15

Figure 16 Example for Normalization: The Company Table.

attributes in any one of these superkeys, you would be able to say which row of the table they occurred in. Note that the last two contain "extra" attributes that are not needed for the functional dependency. Superkeys without such unneeded attributes are called candidate keys. That is, candidate keys are minimal superkeys, sets in which all the attributes are needed for functional dependency. The candidate keys for our example are {SSN, projID}, {empID,projID}, {SSN, projName}, and {empID, projName}. The database designer chooses one of the candidate keys to be the primary key of the relation. All the attributes of any candidate key are called prime attributes.

B. Anomalies

A primary reason to normalize tables is to avoid anomalies that can arise when we update, insert, or delete tuples in a relation. For example, consider the Company database shown in Figure 16 and assume that the primary key is chosen to be {empID, projID}. An update anomaly could occur if we changed information in one tuple, making it inconsistent with the same data in another tuple. For example, if director of the Atlantic project changes from Quinn to Smith, we might update it in the first record, for employee 101, and neglect to update it in other records having the same project. An insertion anomaly occurs when we are unable to insert data because we do not know the value of one of the attributes of the primary key. If there is a new employee who does not yet have a project assignment, we cannot insert the employee's record because the primary key includes projID, for which we have no value. The same problem would arise if we had a project to which no employee were assigned yet. A deletion anomaly occurs when the deletion of one or more records causes us to lose other information as a side effect. For example, if we delete the record for employee 103 we lose all information about the Arctic project, since that employee is the only one assigned to that project. Normalization addresses the issue of anomalies by separating data appropriately.

C. First Normal Form

A relation is in first normal form (abbreviated 1NF) if the domains of all its attributes are atomic, and the value of any attribute in a tuple is a single value from its domain. An equivalent definition is that each attribute is nondecomposable and is functionally dependent on the key. Values that are composites, or multiple values for an attribute, are disallowed. This requirement is actually part of the definition of a relation. A relation that violates this rule is said to be unnormalized. Unnormalized relations are permitted in object-oriented models and in object-relational models, but not in strictly relational ones. The Company table of Figure 16 demonstrates this requirement. Employee 101 is listed twice, once for each project he or she is assigned to. If the table were not in first normal form, we could have put both projects in the same record.

D. Second Normal Form

The definition of second normal form involves the notion of full functional dependency. An attribute or set of attributes B is fully functionally dependent on an attribute or set of attributes A if it is functionally dependent on A, but not on any (proper) subset of A. If the determinant A consists of several attributes, A1, A2, . . . , An and we remove any one of them from A, then B is no longer functionally dependent on this smaller set of attributes. That is, all of the attributes of A are needed to functionally determine B. A relation is in second normal form if it is in first normal form and every nonprime attribute is fully function-

ally dependent on the key. In actuality, the definition extends to full functional dependency on any candidate key as well.

To understand why second normal form is desirable, note that even though the Company relation is in first normal form, we saw that it has insertion, update, and deletion anomalies. Codd identified the source of such problems as partial functional dependencies. The key of the Company relation is {empID, projID}. However, some of the nonkey attributes are functionally dependent on only one of the attributes in the key. For example, empID → empName, without projID. This is an example of partial dependency, which second normal form does not allow. To remedy the problem, the relation should be decomposed by relational algebra projection into three relations, each of which is in second normal form. They are

Employee(empID, SSN, empName, deptID, deptMgr, jobTitle, salary)

Project(projID, projName, projDir, budget)

Assignment(empID, projID, hoursAssigned, rating)

as shown in Figure 17. In this schema, each attribute of Employee stores a single fact about one employee, each attribute of Project a single fact about one project, and each attribute of Assignment a single fact about one employee's work on one project. Note that the insertion, deletion, and update anomalies are resolved in this decomposition, and all the constraints and facts in the original table are preserved.

E. Third Normal Form

Third normal form is based on the notion of transitive dependency. If A, B, and C are attributes of relation R, such that A → B, and B → C, then C is transitively dependent on A, unless either B or C is a candidate key or part of a candidate key. Equivalently, a transitive dependency exists when a nonprime attribute determines another nonprime attribute. A relation is in third normal form if it is in second normal form and no nonprime attribute is transitively dependent on any candidate key. In the Employee table of the company database in Figure 17, we see that deptID → deptMgr, so we have one nonprime attribute determining another. The only candidate keys for the relation are empID and SSN. This means that deptMgr is transitively dependent on empID, the primary key, since neither deptID nor deptMgr is part of any candidate key for this relation. Because of the transitive dependency, we can still have anomalies. If we have a new department with a new manager, we cannot insert this information unless we have an employee, which is an insertion anomaly. If the manager of department 10 changes from Munez to Miller, we may update one employee's record and fail to update another's in the same department, leading to an update anomaly. If we delete the record of the only employee in a department, we lose the information about the department, including the manager's name.

To correct the design, we can decompose the Employee table into

Employee1(empID, SSN, empName, *deptID,* jobTitle, salary)

Department(deptID, deptMgr)

resulting in the schema shown in Figure 18. The Project and Assignment tables are already in third normal form. In the Project table, projID and projName are both candidate keys, and there are no transitive dependencies. In Assignment, only the composite {empID, projID} is a candidate key, and there are no other functional dependencies.

F. Boyce–Codd Normal Form

Boyce–Codd normal form is slightly stricter than 3NF. A relation is in Boyce–Codd normal form (BCNF) if and only if every determinant is a candidate key. (Recall that 3NF allows a determinant to be a part of a candidate key and does not require it to be an entire candidate key.) It is rare to find a relation that is 3NF without also being BCNF, but it is possible when a relation has composite candidate keys that have at least one attribute in common. Our revised table, Em-

Employee(empID, SSN, empName, deptID, deptMgr, jobTitle, salary)

Project(projID, projName, projDir, budget)

Assignment(*empID, projID,* hoursAssigned, rating)

Figure 17 The Company Database in 2NF.

Employee1(empID, SSN, empName, *deptID,* jobTitle, salary)

Department(deptID, deptMgr)

Project(projID, projName, projDir, budget)

Assignment(*empID, projID,* hoursAssigned, rating)

Figure 18 The Company Database in 3NF and BCNF.

ployee1, is in BCNF because it has only two determinants, empID and SSN, both of which are candidate keys. All of the other tables in the schema shown in Figure 18 are also in BCNF. Although a schema can always be decomposed by projection into BCNF relations, it is not always desirable to do so, because the decomposition may place a determinant and the attribute(s) it determines in different relations, thus losing a functional dependency. It has been demonstrated that while it is always possible to find a 3NF decomposition that preserves functional dependencies, it is not always possible to find a BCNF one that does so.

G. Multivalued Dependencies and Fourth Normal Form

Although BCNF eliminates anomalies due to functional dependencies, Fagin, Zaniolo, and Delobel independently identified another type of dependency that can cause problems due to repetition of data. Multivalued dependency may arise out of the process of normalization to achieve first normal form. Recall that first normal form forbids multiple values in a cell of a table. One solution is to repeat the rest of the data along with each of the values of the cell, making the multivalued attribute part of the key. If we have two attributes that are, by nature, multivalued, we must create tuples for every combination of values of one with values of the second. A multivalued dependency exists when there are three attributes A, B, and C in a relation R such that for each value of A the set of B values associated with the A value are independent of the set of C values associated with the A value. We say A multidetermines B and A multidetermines C. By definition, multivalued dependencies occur in pairs. We write

$A \twoheadrightarrow B$

$A \twoheadrightarrow C.$

A trivial multivalued dependency A - >> B is one where either B is a subset of A, or A and B together make up all the attributes of R. A relation is in fourth normal form if it is in BCNF and has no nontrivial multivalued dependencies. For example, consider the relation

Emp(empID, skill, dependentName).

We will assume an employee can have several skills and several dependents. An unnormalized instance of this relation is shown in Figure 19. Since this is not a valid 1NF table, we need to normalize it by removing the repeating values from the skill and dependents cells. When we "flatten" the table in this way, we have to repeat all combinations of skill and dependent name for each employee, to avoid the appearance of a relationship between the skill and the dependent name. The resulting table instance is shown in Figure 20. In this table, all three attributes form the key, and we have

empID	skill	dependentName
E101	French	Mary
	Data entry	Tom, Jr.
E105	Systems analysis	John
	Database design	Jill

Figure 19 The Unnormalized Emp Table.

empID $\twoheadrightarrow$ skill

empID $\twoheadrightarrow$ dependentName.

The table is not in 4NF. To make it 4NF, we decompose it into two tables, as shown in Figure 21:

Emp-skill(<u>empID, skill</u>)

Emp-dependent(<u>empID, dependent</u>).

Each of these tables is in 4NF.

H. Lossless Decomposition and Fifth Normal Form

The process of normalization involves examining tables for dependencies and then decomposing them by projection. However, the decomposition cannot be done arbitrarily. For each of the projections we have created in previous examples, it is possible to get the original table back, with exactly its original tuples, by joining the decomposed tables. A projection that can

empID	skill	dependentName
E101	French	Mary
E101	French	Tom, Jr.
E101	Data entry	Mary
E101	Data entry	Tom, Jr.
E105	Systems analysis	John
E105	Systems analysis	Jill
E105	Database design	John
E105	Database design	Jill

Figure 20 The Emp Table in First Normal Form.

Emp-skill:

empID	skill
E101	French
E101	Data entry
E105	Systems analysis
E105	Database design

Emp-dep:

empID	dependentName
E101	Mary
E101	Tom, Jr.
E105	John
E105	Jill

Figure 21 The Emp Database in 4NF.

(a) empID	(a) task#
e10	t1
e10	t2
e12	t1

(b) task#	(b) proj#
t1	p200
t2	p100
t1	p100

(c) empID	(c) proj#
e10	p200
e10	p100
e12	p100

Figure 23 Projections of EmpTaskProj.

be reversed through a join to recreate the original relation is called a lossless decomposition, or lossless join. Not all decompositions are lossless, which means there are projections whose join does not equal the original relation. As an example of a lossy projection and a join dependency, consider the relation EmpTaskProj(empID, task#, proj#) shown in Figure 22. The table shows which employees perform which tasks for which projects. We can decompose the table by projection into tables (a) and (b) of Figure 23. (For the present, we ignore table (c) of Figure 23.) However, when we join those two tables, in Figure 24, we do not get the original table back, which means that this is a lossy projection. We see that we get an extra tuple that did not appear in the original table. This is an example of a spurious tuple, one created by the projection and join processes. Instead of representing real data, it is an artifact of the decomposition process. The original table can be recreated by joining the result with table (c) of Figure 23, as shown in Figure 25.

A join dependency exists when for a relation R with subsets of its attributes A, B, . . . , Z, R is equal to the join of its projections on A, B, . . . , Z. A relation is in fifth normal form if every join dependency is implied by the candidate keys. This means that the only valid decompositions are those involving the candidate keys. Beyond those, there is no advantage to be had by decomposing the relation further. Join dependencies are a generalization of multivalued dependencies and they can be quite subtle, making it difficult to find them. They are believed to be relatively rare, so in practice designers often stop at 4NF or even BCNF or 3NF, especially if the designer wishes to preserve functional dependencies.

VII. OBJECT-RELATIONAL DATABASES

In recent years, the paradigm for programming languages has shifted to object orientation, with languages such as C++ and Java increasing in popularity. Data to be stored in databases have become increasingly complex, as structured data, multimedia, and other types of information have become more common. These trends have resulted in changes to relational database management systems, creating a merging of relational and object-oriented systems, often referred to as object-relational databases. Many relational DBMSs such as Oracle now support some object-oriented features. Since 1999, the SQL3 standard has also included some object features.

One of the first modifications to the strict relational model was the development of "nested relations." In addition to atomic domains, the object-relational model allows some user-defined data types, including structured types. For example, a database designer might define custAddress to consist of street, city, state, and zip code. Similarly, custName might be defined to consist of firstName, lastName. Then a Customer table could be defined to use custName

empID	task#	proj#
e10	t1	p200
e10	t2	p100
e10	t1	p100
e12	t1	p100

Figure 22 Original Table EmpTaskProj.

empID	task#	proj#
e10	t1	p200
e10	t1	p100
e10	t2	p100
e12	t1	p200
e12	t1	p100

Figure 24 First Join using Table a and Table b of Figure 23.

empID	task#	proj#
e10	t1	p200
e10	t1	p100
e10	t2	p100
e12	t1	j200

Figure 25 Join of First Join with Table c of Figure 23 over {empID, proj#}.

and custAddress as attributes, resulting in a schema that contains schemas, which we could write as

Customer(custID, custName(firstName, lastName), custAdd(street, city, state, zipcode), creditRating).

SQL3 also permits attributes to contain arrays of values. For example, in the Item table items might be available in several colors, and the colors might be stored in an array. Tables that contain such structured values or multiple values would not satisfy the traditional definition of first normal form.

User-defined data types can have their own methods, which are special functions defined for members of the type. Structured types can participate in type hierarchies, with inheritance of attributes and methods by the subtype. The tuples of tables may have unique identifiers that are independent of the values of their attributes. Among the data types permitted is LOB, Large Object, a type that can be used to store multimedia objects. Reference types are also permitted, allowing tuples to refer to other tuples.

VIII. CODD'S RULES FOR RELATIONAL DATABASES

Codd proposed rules that a database management system should follow to be considered relational. Although commercial DBMSs, especially those that are object-relational, do not follow all of these rules, it is interesting to examine the standards that Codd proposed.

1. Information representation. All information is represented at the logical level only as values in tables.
2. Guaranteed access. Users must be able to access any data item by providing its table name, primary key value, and column name.
3. Treatment of null values. Null values (distinct from zero or the empty string) must be represented systematically, regardless of data type.
4. Relational catalog. The system catalog is represented relationally and can be queried in the same way as stored data.
5. Comprehensive data sublanguage. Although a relational system can provide other languages, it must have at least one language that supports data and view definition, data manipulation, integrity constraints, user authorization, and transaction management.
6. View updates. All views that are theoretically updatable in the relational model must be actually updatable in the system.
7. High-level update operations. Any base relation or derived relation that can be treated as a single operand for retrieval can also be handled as a single operand for insertion, update, and deletion operations.
8. Physical data independence. Application programs are not affected by changes to storage representation or access methods.
9. Logical data independence. Application programs are not affected by changes at the logical level that do not affect the information content.
10. Integrity rules. Integrity constraints must be expressed in the data sublanguage and stored in the system catalog, rather than in application programs.
11. Distribution independence. Regardless of whether the data are distributed or centralized, the data manipulation language commands used by applications and users must remain logically the same.
12. Nonsubversion. If the system has a low-level language that supports record-at-a-time access, that language cannot be used to bypass integrity constraints expressed in the high-level language.

The overarching rule, which Codd called Rule 0, or the foundation rule, is that any system that is described as relational must manage its stored data using only its relational capabilities.

SEE ALSO THE FOLLOWING ARTICLES

Database Administration • Database Development Process • Database Systems • Data Modeling: Entity-Relationship Data Model • Data Modeling: Object-Oriented Data Model • Network Database Systems • Object-Oriented Databases • Structured Query Language (SQL) • Temporal Data Model and Query Language Concepts

BIBLIOGRAPHY

Chamberlin, D. (1976). Relational data-base management systems. *ACM Computing Surveys,* Vol. 8, No. 1, 43–66.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the Association for Computing Machinery,* Vol. 13, No. 6, 377–387.

Codd, E. F. (1982). The 1981 ACM Turing award lecture: Relational database: A practical foundation for productivity. *Communications of the ACM,* Vol. 25, No. 2, 109–117.

Codd, E. F. (1985). Is your DBMS really relational? *Computerworld,* October 14, 1–9.

Date, C. J. (2002). *An introduction to database systems,* 7th ed. Reading, MA: Addison–Wesley.

Delobel, C. (1978). Normalization and hierarchical dependencies in the relational data model. *ACM Transactions on Database Systems,* Vol. 3, No. 3, 201–222.

Elmasri, R., and Navathe, S. (2002). *Fundamentals of database systems,* 3rd ed. Reading, MA: Addison–Wesley.

Fagin, R. (1977). Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Sysems,* Vol. 2, No. 3, 262–278.

Garcia-Molina, H., Ullman, J., and Widom, J. (2002). *Database systems: The complete book.* Upper Saddle River, NJ: Prentice Hall.

Maier, D. (1983). *The theory of relational databases.* Rockville, MD: Computer Science Press.

Ramakrishnan, R., and Gehrke, J. (2000). *Database management systems,* 2nd ed. New York: McGraw–Hill.

Stonebraker, M. (Ed.). (1986). *The INGRES papers.* Reading, MA: Addison–Wesley.

Todd, S. (1976). The Peterlee relational test vehicle—A system overview. *IBM Systems Journal,* Vol. 15, No. 4, 285–308.

Ullman, J. (1988). *Principles of database and knowledge-base systems,* Vol. I. Rockville, MD: Computer Science Press.

Ullman, J. (1989). *Principles of database and knowledge-base systems,* Vol. II. Rockville, MD: Computer Science Press.

Zaniolo, C., and Melkanoff, M. (1981). On the design of relational database schemata. *ACM Transactions on Database Systems,* Vol. 6, No. 1, 1–47.