# RELATIONAL DATABASES

Managing a large amount of persistent data with computers requires storing and retrieving these data in files. However, it was found in the early 1960s that files are not sufficient for the design and use of more and more sophisticated applications. As a consequence, database systems have become a very important tool for many applications over the past 30 years. Database management systems (DBMSs) aim to provide users with an efficient tool for good modeling and for easy and efficient manipulation of data. It is important to note that concurrency control, data confidentiality, and recovery from failure are also important services a DBMS should offer. The very first DBMSs, known as hierarchical and then network systems, were based on a hierarchical and then network-like conceptual data organization, which actually reflects the physical organization of the underlying files. Thus, these systems do not distinguish clearly between the physical and the conceptual levels of data organization. Therefore, these systems, although efficient, have some important drawbacks, among which we mention data redundancies (which should be avoided) and a procedural way of data manipulation, which is considered not easy enough to use.

The relational model, proposed by Codd in 1970 (1), avoids the drawbacks mentioned previously by distinguishing explicitly between the physical and conceptual levels of data organization. This basic property of the relational model is a consequence of the fact that, in this model, users see the data as tables and do not have to be aware how these tables are physically stored. The tables of a relational database are accessed and manipulated as a whole, contrary to languages based on hierarchical or network models, according to which data are manipulated on a record-by-record basis. As a consequence, data manipulation languages for relational databases are set-oriented and so, fall into the category of declarative languages, in which there is no need of control structures, such as conditional or iterative statements. On the other hand, because relationships are a well-known mathematical concept, the relational model stimulated a lot of theoretical research, which led to successful implementations. As an example of a relational database, Fig. 1 shows the two tables, called EMP and DEPT, of a sample database for a business application.

The main results obtained so far are summarized as follows:

1. The expressional power of relational data manipulation languages is almost that of first-order logic without functional symbols. Moreover, relational languages have large capabilities of optimization. This point is of particular importance, because it guarantees that data are efficiently retrieved, independently of the way the query is issued by the user.
2. Integrity constraints, whose role is to account for properties of data are considered within the model. The most important and familiar are the functional dependencies. Research on this topic led to theoretical criteria for what is meant by a "good" conceptual data organization for a given application.
3. A theory of concurrency control and transaction management has been proposed to account for the dynamic aspects of data manipulation with integrity constraints. Research in this area led to actual methods and algorithms which guarantees that, in the presence of multiple updates in a multiuser environment, the modified database still satisfies the integrity constraints imposed on it.

These fundamental aspects led to actual relational systems that rapidly acquired their position in the software market and still continue to do so today. Relational DBMSs are currently the key piece of software in most business applications running on various types of computers, ranging from mainframe systems to personal computers (PCs). Among the relational systems available on the marketplace, we mention DB2 (IBM), INGRES (developed at the University of California, Berkeley), and ORACLE (Oracle Corp.), all of which implement the relational model of databases together with tools for developing applications.

| EMP | *empno* | *ename* | *sal* | *deptno* |
|---|---|---|---|---|
| | 123 | john | 23,000 | 1 |
| | 234 | julia | 50,000 | 1 |
| | 345 | peter | 7,500 | 2 |
| | 456 | laura | 12,000 | 2 |
| | 567 | paul | 8,000 | 1 |

| DEPT | *deptno* | *dname* | *mgr* |
|---|---|---|---|
| | 1 | sales | 234 |
| | 2 | staff | 345 |

**Figure 1.** A sample relational database D.

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 1999 John Wiley & Sons, Inc.

In the remainder of this article, we focus on the theory of the relational model and on basic aspects of dependency theory. Then, we deal with problems related to updates and transaction management, and we briefly describe the structure of relational systems and the associated reference language called SQL. We conclude with a brief discussion on extensions of the relational model currently under investigation.

## THEORETICAL BACKGROUND OF RELATIONAL DATABASES

The theory of the relational model of databases is based on relationships. Although relationships are well known in mathematics, their use in the field of databases requires definitions that slightly differ from those usual in mathematics. Based on these definitions, basic operations on relationships constitute relational algebra, which is closely related to first-order logic. Indeed, relational algebra has the same expressional power as a first-order logic language, called relational calculus, and this relationship constitutes the basis of the definition of actual data manipulation languages, among which the language called SQL is now the reference.

### Basic Definitions and Notations

The formal definition of relational databases starts with a finite set, called the *universe,* whose elements are called *attributes.* If U denotes a universe, each attribute A of U is associated with a nonempty and possibly infinite set of value (or constants), called the *domain of* A and denoted by *dom*(A). Every nonempty subset of U is called a *relation scheme* and is denoted by the juxtaposition of its elements. For example, in the database of Fig. 1, the universe U contains the attributes *empno, ename, sal, deptno, dname* and *mgr* standing respectively for: numbers, names and salaries of employees, numbers, names of departments, and numbers of managers. Moreover, we consider here that *empno, deptno,* and *mgr* have the same domain, namely the set of all positive integers, whereas the domain of the attributes *ename* and *dname* is the set of strings of alphabetic characters of length at most 10.

Given a relationship scheme R, a *tuple t over* R is a mapping from R to the union of the domains of the attributes in R, so that, for every attribute A in R, *t*(A) is an element of *dom*(A). Moreover, if R′ is a nonempty subset of R the *restriction of t to* R′, being the restriction of a mapping, is also a tuple, denoted by *t*.R′. As a notational convenience, tuples are denoted by the juxtaposition of their values, assuming that the order in which values are written corresponds to the order in which attributes in R are considered.

Given a universe U and a relation scheme R, a *relation over* R a is a *finite* set of tuples over R, and a *database over* U is a set of relations over relations schemes obtained from U.

### Relational Algebra

From a theoretical point of view, querying a database consists of computing a relation (which in practice is displayed as the answer to the query) based on the relation in the database. The relation to be computed can be expressed in two different languages: relational algebra, which explicitly manipulates relation, and relational calculus, which is based on first-order logic. Roughly speaking, relational calculus is the *declarative* counterpart of relational algebra, seen as a *procedural* language.

The six fundamental operations of relational algebra are union, difference, projection, selection, join, and renaming (note that replacing the join operation by the cartesian-product is another popular choice discussed in Refs. 2 and 3). The formal definitions of these operations are as follows: Let $r$ and $s$ be two relations over relation schemes $R$ and $S$, respectively. Then

1. *Union.* If R = S then $r \cup s$ is a relation defined over R, such that $r \cup s = \{t \mid t \in r \text{ or } t \in s\}$. Otherwise $r \cup s$ is undefined.
2. *Difference.* If R = S then $r - s$ is a relation defined over R, such that $r - s = \{t \mid t \in r \text{ and } t \notin s\}$. Otherwise $r - s$ is undefined.
3. *Projection.* Let Y be a relation scheme. If Y ⊆ R, then $\pi_Y(r)$ is a relation defined over Y, such that $\pi_Y(r) = \{t \mid \exists \mu \in r \text{ such that } u.Y = t\}$. Otherwise $\pi_Y(r)$ is undefined.
4. *Selection* of $r$ with respect to a condition $C$: $\sigma_c(r)$ is a relation defined over R, such that $\sigma_c(r)\,) = \{t \mid t \in r$ and $t$ satisfies $C\}$. Selection conditions are either atomic conditions or conditions obtained by combination of atomic conditions, using the logical connectives $\vee$ (or), $\wedge$ (and), or $\neg$ (not). An atomic condition is an expression of the form A Θ A′ or A Θ $a$ where A and A′ are attributes in R whose domains are “compatible” [i.e., it makes sense to compare a value in *dom*(A) with a value in *dom*(A′)], $a$ is a constant in *dom*(A), and Θ is an operator of comparison, such as $<$, $>$, $\leq$, $\geq$ or $=$.
5. *Join.* $r \bowtie s$ is a relation defined over $R \cup S$, such that $r \bowtie s = \{t \mid t.R \in r \text{ and } t.S \in s\}$.
6. *Renaming.* If A is an attribute in R and B is an attribute not in R, such that *dom*(A) = *dom*(B), then $\rho_{B \leftarrow A}(r)$ is a relation defined over (R − {A}) ∪ {B} whose tuples are the same as those in $r$.

For example, in the database of Fig. 1, the following expression computes the numbers and names of all departments having an employee whose salary is less than 10,000:

$$\mathbf{E} : \pi_{\text{deptno dname}}[\sigma_{\text{sal}<10,000}(\text{EMP} \bowtie \text{DEPT})].$$

Figure 2 shows the steps for evaluating this expression against the database of Fig. 1. As an example of using renaming, the following expression computes the numbers of employees working in at least two different departments:

$$\mathbf{E_1} : \pi_{\text{empno}}\big[\sigma_{\text{deptno}\neq\text{dnumber}}(\text{EMP}) \bowtie \rho_{\text{dnumber}\leftarrow\text{deptno}}[\pi_{\text{deptno empno}}(\text{EMP})]\big].$$

The operations introduced previously enjoy properties, such as commutativity, associativity, and distributivity [see (3) for full details]. The properties of the relational operators allow for syntactic transformations according to which the same result is obtained, but through a more efficient computation. For instance, instead of evaluating the previous expression **E**, it is more efficient to consider the following expression:

$$\mathbf{E}' : \pi_{\text{deptno dname}}[\sigma_{\text{sal}<10,000}(\text{EMP}) \bowtie \pi_{\text{deptno dname}}(\text{DEPT})].$$

(*a*) EMP ⋈ DEPT

| *empno* | *ename* | *sal* | *deptno* | *dname* | *mgr* |
|---|---|---|---|---|---|
| 123 | john | 23,000 | 1 | sales | 234 |
| 234 | julia | 50,000 | 1 | sales | 234 |
| 345 | peter | 7,500 | 2 | staff | 345 |
| 456 | laura | 12,000 | 2 | staff | 345 |
| 578 | paul | 8,000 | 1 | sales | 234 |

(*b*) $\sigma_{sal<10,000}$ (EMP ⋈ DEPT)

| *empno* | *ename* | *sal* | *deptno* | *dname* | *mgr* |
|---|---|---|---|---|---|
| 345 | peter | 7,500 | 2 | staff | 345 |
| 578 | paul | 8,000 | 1 | sales | 234 |

(*c*) $\pi_{deptno\ dname}[\sigma_{sal<10,000}$ (EMP ⋈ DEPT)]

| *deptno* | *dname* |
|---|---|
| 2 | staff |
| 1 | sales |

**Figure 2.** The intermediate relations in the computation of expression **E** applied to the database D of Fig. 1. (a) the computation of the join; (b) the computation of the selection; and (c) the computation of the projection.

Indeed, the intermediate relations computed for this expression are "smaller" than those of Fig. 2 in the number of rows and the number of columns.

Such a transformation is known as query optimization. To optimize an expression of relational algebra, the expression is represented as a tree in which the internal nodes are labeled by operators and the leaves are labeled by the names of the relations of the database. Optimizing an expression consists of applying properties of relational operators to transform the associated tree into another tree for which the evaluation is more efficient. For instance, one of the most frequent transformations consists of pushing down selections in the tree to reduce the number of rows of intermediate relationships. We refer to (2) for a complete discussion of query optimization techniques and for any topic related to databases. Although efficient in practice, query optimization techniques are not optimal, because, as Kanellakis notices (4), the problem of deciding whether two expressions of relational algebra always yield the same result is impossible to solve.

### Relational Calculus

The existence of different ways of expressing a given query in relational algebra stresses the fact that, as mentioned previously, it is a procedural language. Fortunately, relational algebra has a declarative counterpart, relational calculus. This comes from the observation that, if $r$ is a relation defined over a relation scheme R containing $n$ distinct attributes, then membership of a given tuple $t$ in $r$ is equivalently expressed by first-order formalism if we regard $r$ as an $n$-ary predicate, and $t$ as an $n$-ary vector of constants and if we state that the atomic formula $r(t)$ is true. More formally, the correspondence between relational algebra and calculus is as follows: Given a database D = $\{r_1, r_2, \ldots, r_n\}$ over a universe U and with schema $\{R_1, R_2, \ldots, R_n\}$, we consider a first-order alphabet with the usual connectives ($\wedge, \vee, \neg$) and quantifiers ($\exists, \forall$) where

1. the set of constant symbols is the union of all domains of the attributes in U:
2. the set of predicate symbols is $\{r_1, r_2, \ldots, r_n\}$, where each $r_i$ is a predicate symbol whose arity is the cardinality of $R_i$; and
3. the variable symbols may range over tuples, in which case the language is called *tuple* calculus, or over domain elements, in which case the language is called *domain* calculus.

One should notice that no function symbols are considered in relational calculus. Based on such an alphabet, formulas of interest are built up as usual in logic, but with some syntactic restrictions explained later. Now we recall that without loss of generality, a well-formed formula has the form $\Psi = (Q_1)(Q_2) \ldots (Q_k)[\varphi(x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_1)]$ where $x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_1$ are the only variable symbols occurring in $\varphi$, where $(Q_i)$ stands for $(\exists x_i)$ or $(\forall x_i)$ and where $\varphi$ is a quantifier-free formula built up from connectives and atomic formulas (atomic formulas have the form $r(t_1, t_2, \ldots, t_n)$ where $r$ is an $n$-ary predicate symbol and $t_j$ is either a variable or a constant symbol). Moreover, in the formula $\Psi$, the variables $x_i$ are bound (or quantified) and the variables $y_j$ are free (or not quantified). See (5) for full details on this topic.

In the formalism of tuple calculus, the relational expression **E** is written as

$$\{z|(\exists x)(\exists y)(\mathrm{EMP}(x) \wedge \mathrm{DEPT}(y) \wedge y.deptno{=}z.deptno \\ \wedge y.dname{=}z.dname \\ \wedge x.deptno{=}y.deptno \wedge x.sal < 10,000)\}$$

One should note that, in this formula, variables stand for tuples, whose components are denoted as restrictions in relational algebra. Considering domain calculus, the previous formula is written as follows:

$$\{z_1 z_2|(\exists x_1)(\exists x_2)(\exists x_3)(\exists y_1)(\mathrm{EMP}(x_1, x_2, x_3, z_1) \\ \wedge \mathrm{DEPT}(z_1, z_2, y_1) \wedge x_3 < 10,000)\}$$

where $z_1$ and $z_2$ are free variables ranging, respectively, over all possible numbers and names of departments.

The satisfaction of a formula $\Psi$ in a database D is defined in a standard way, as in first-order logic. In the context of databases, however, some well-formed formulas must be discarded because relations are assumed to be finite, and thus, so must be the set of tuples satisfying a given formula in a database. For instance, the domain calculus formula $(\exists x)[\neg r(x, y)]$ must be discarded, because in any database, the set of constants $a$ satisfying the formula $\neg r(x_0, a)$ for some appropriate $x_0$ may be infinite (remember that domains may be infinite). The notion of safeness is based on what is called the domain of a formula $\Psi$, denoted by DOM($\Psi$). DOM($\Psi$) is defined as the set of all constant symbols occurring in $\Psi$, together with all constant symbols of tuples in relations occurring in $\Psi$ as predicate symbols. Hence, DOM($\Psi$) is a finite set of constants and $\Psi$ is called *safe* if all tuples satisfying it in D contain only constants of DOM($\Psi$). To illustrate the notion of safeness, again consider the formula $\Psi = (\exists x)[\neg r(x, y)]$. Here DOM($\Psi$) = $\{\alpha \mid \alpha$ occurs in a tuple of $r\}$, and so, $\Psi$ may be satisfied in D by values $\beta$ not in DOM($\Psi$). Therefore, $\Psi$ is a nonsafe formula. On the other hand, the formula $\Psi' = (\exists x)[\neg r(x, y) \wedge s(x, y)]$ is safe, because every $\beta$ satisfying $\Psi'$ in D occurs in DOM($\Psi'$).

It is important to note that tuple and domain calculus are equivalent languages that have resulted in the emergence of actual languages for relational systems. A formal proof of the equivalence between relational calculus and relational algebra was given by Codd in Ref. 6.

## DATA DEPENDENCIES

The theory of data dependencies has been motivated by problems of particular practical importance, because in all applications, data stored in a database must be restricted so as to satisfy some required properties or constraints. For instance, in the database of Fig. 1, two such properties could be (1) two departments with distinct names cannot have the same number and (2) a department has only one manager, so that the relation DEPT cannot contain two distinct tuples with the same *deptno* value. Investigations on constraints in databases have been carried out in the context of the relational model in order to provide sound methods for the design of database schemas. The impact of constraints on schema design is exemplified through properties (1) and (2). Indeed, assume that the database consists of only one relation defined over the full universe. Then clearly, information about a given department is stored as many times as the number of its employees, which is redundant. This problem has been solved by the introducing normal forms in the case of particular dependencies called functional dependencies. On the other hand, another problem that arises in the context of our example is the following: assuming that a database D satisfies the constraints (1) and (2), does D satisfy other constraints? Clearly, this problem, called the implication problem, has to be solved to make sure that all constraints are considered at the design phase just mentioned. Again, the implication problem has been solved in the context of functional dependencies. In what follows, we focus on functional dependencies, and then, we outline other kinds of dependencies that have also been the subject of research.

### The Theory of Functional Dependencies

Let $r$ be a relation over a relation scheme R, and let X and Y be two subschemes of R. The functional dependency from X to Y, denoted by $X \rightarrow Y$, is satisfied by $r$ if, for all tuples $t$ and $t'$ in $r$, the following holds: $t.X = t'.X \Rightarrow t.Y = t'.Y$. Then, given a set F of functional dependencies and a dependency $X \rightarrow Y$, F implies $X \rightarrow Y$ if every relation satisfying the dependencies in F also satisfies the dependency $X \rightarrow Y$. For instance, for R = (A, B, C) and F = ($A \rightarrow B$, $AB \rightarrow C$), it can be seen that F implies $A \rightarrow C$. However, this definition of the implication of functional dependencies is not effective from a computational point of view. An axiomatization of this problem, proposed in (7), consists of the following rules, where X, Y, and Z are relation schemes:

1. $Y \subseteq X \Rightarrow X \rightarrow Y$
2. $X \rightarrow Y \Rightarrow XZ \rightarrow YZ$
3. $X \rightarrow Y, Y \rightarrow Z \Rightarrow X \rightarrow Z$

A derivation using these axioms is defined as follows: F derives $X \rightarrow Y$ if either $X \rightarrow Y$ is in F or $X \rightarrow Y$ can be generated from F using repeatedly the axioms above. Then, the soundness and completeness of these axioms is expressed as follows: F implies $X \rightarrow Y$ if and only if F derives $X \rightarrow Y$, thus providing an effective way for solving the implication problem in this case.

An important aspect of functional dependencies is that they allow for the definition of normal forms which characterize suitable database schemas. Normal forms are based on the notion of key defined as follows: if R is a relation scheme with functional dependencies F, then K is a key of (R, F) if K is a minimal relation scheme with respect to set inclusion such that F implies (or derives) $K \rightarrow R$. Four normal forms can be defined, among which we mention here only three of them:

1. The first normal form (INF) stipulates that attributes are atomic in the relational model. This is implicit in the definitions of relational databases but restricts the range of applications that can easily been taken into account. This explains, in particular, the emergence of object-oriented models of databases.
2. The third normal form (3NF) stipulates that attributes participating in no keys depend fully and exclusively on keys. The formal definition is as follows: (R, F) is in 3NF if, for every derived dependency $X \rightarrow A$ from F, such that A is an attribute not in X and appearing in no keys of (R, F), X contains a key of (R, F).
3. The Boyce–Codd normal form (BCNF), is defined as the previous form, except that the attribute A may now appear in a key of (R, F). Thus, the formal definition is the following: (R, F) is in BCNF if, for every derived dependency $X \rightarrow A$ from F, such that A is an attribute not in X, X contains a key of (R, F).

It turns out that every scheme (R, F) in BCNF is in 3NF, whereas the contrary is false in general. Moreover, 3NF and BCNF characterize those schemes recognized as suitable in practice. If a scheme (R, F) is neither 3NF nor BCNF, then it is always possible to decompose (R, F) into subschemes that are at least 3NF. More precisely, by schema decomposition, we mean the replacement of (R, F) by schemes ($R_1$, $F_1$), ($R_2$, $F_2$), . . ., ($R_k$, $F_k$), where

1. each $R_i$ is a subset of R and R in the union of the $R_i$s;
2. each $F_i$ is the set of all dependencies $X \rightarrow Y$ derivable from F, such that $XY \subseteq R_i$; and
3. each ($R_i$, $F_i$) is in 3NF or in BCNF.

Furthermore, this replacement must ensure that data and dependencies are preserved in the following sense:

1. Data preservation: starting with a relation $r$ which satisfies F, the relations $r_i$ are the projections of $r$ over $R_i$, and their join must be equal to $r$.
2. Dependency preservation: the set F and the union of the sets $F_i$ must derive exactly the same functional dependencies.

In the context of functional dependencies, data preservation is characterized as follows, in the case where $k = 2$: the decomposition of (R, F) into ($R_1$, $F_1$), ($R_2$, $F_2$) preserves the data if F derives at least one of the two functional dependencies $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$. If k is greater than 2, then the previous result can be generalized, using properties of the join operator. Unfortunately, no such easy to check property is known for dependency preservation. What has to be done in practice is to make sure that every dependency of F can be derived from the union of the $F_i$s.

It has been shown that it is always possible to decompose a scheme (U, F) so that data and dependencies are preserved and the schemes ($R_i$, $F_i$) are all at least in 3NF. But it should be noticed that BCNF is not guaranteed when decomposing a relation scheme. Two kinds of algorithms have been implemented for schema decomposition: the synthesis algorithms (which generate the schemes based on a canonical form of the dependencies of F) and the decomposition algorithms (which repeatedly split the universe U into two subschemes). Synthesis algorithms ensure data and dependency preservation together with schemes in 3NF (at least), whereas decomposition algorithms ensure data preservation together with schemes in BCNF, but at the cost of a possible loss of dependencies.

### More on Data Dependencies

Dependencies other than functional dependencies have been widely studied in the past. In particular, multivalued dependencies and their interaction with functional dependencies have motivated much research. The intuitive idea behind multivalued dependencies is that, in a relation over R, a value over X is associated with a set of values over Y, and is independent of the values over R − XY. An example of multivalued dependencies is the following: assume that we have R = {*empno, childname, car*}, to store the names of the children and the cars of employees. Clearly, every *empno* value is associated with a fixed set of names (of children), independent of the associated *car* values. Multivalued dependencies and functional dependencies have been axiomatized soundly and completely, which has led to an additional normal form, called the fourth normal form, and defined similarly to BCNF.

Other dependencies of practical interest which have been studied are inclusion dependencies. For example, in the database of Fig. 1, we may like to state that every manager is an employee, which is expressed as follows: $\pi_{mgr}(DEPT) \subseteq \pi_{empo}(EMP)$. In general, an inclusion dependency is $\pi_X(r) \subseteq \pi_Y(s)$ where $r$ and $s$ are relations of the database and where X and Y are relation schemes, such that the projections and the inclusion are defined. Although it has been shown that the implication problem for inclusion dependencies in the presence of functional dependencies is not decidable [see (2)], a restricted case of practical significance is decidable in polynomial time: the restriction is roughly that the relations in inclusion dependencies are all unary.

## DATABASE UPDATES

Although updates are an important issue in databases, this area has received less attention from the research community than the topics just addressed. Roughly speaking, updates are basic insert, delete, or modify operations defined on relations seen as *physical* structures, and no theoretical background similar to that discussed for queries is available for updates. As a consequence, no declarative way of considering updates has been proposed so far, although there is much effort in this direction. Actually, current relational systems handle sophisticated updates procedurally, based on the notion of *transactions,* which are programs containing update statements. An important point is that, to maintain data consistency, these programs must be considered as units, in the sense that either all or none of their statements are executed. For instance, if a failure occurs during the execution of a transaction, all updates performed before the failure must be undone before rerunning the whole program. In what follows, we first discuss the relationship between updates and data dependencies, and then, we give a short introduction to transaction execution.

### Updates and Data Dependencies

There are two main ways to maintain the database consistent with respect to constraints in the presence of updates: (1) reject all updates contradicting a constraint; (2) take appropriate actions to restore consistency with respect to constraints. To illustrate these two ways of treating updates, let us consider again the database of Fig. 1 and let us assume that the relation DEPT must satisfy the functional dependency *deptno* → *dname mgr.* According to (1) previous, the insertion in DEPT of the tuple 1 *toy* 456 is rejected, whereas it is accepted according to (2) previous, if, in addition, the tuple 1 *sales* 234 is removed from DEPT. Actually, it turns out that (1) gives priority to "old" knowledge over "new" knowledge, whereas (2) does the opposite. Clearly, updating a database according to (1) or (2) depends on the application. In practice, policy (1) is implemented as such for keys and policy (2) is specified by transactions.

Before we come to problems related to transaction execution, we would like to mention that an important and emerging issue related to policy (2) is that of *active rules.* This new concept is considered the declarative counterpart of transactions, and thus, is meant as an efficient tool to specify how the database should react to updates, or, more generally, to events.

Active rules are rules of the form: **on** ⟨event⟩ **if** ⟨condition⟩ **then** ⟨action⟩, and provide a declarative formalism for ensuring that data dependencies remain satisfied in the presence of updates. For example, if we consider the database of Fig. 1 and the inclusion dependency $\pi_{mgr}(DEPT) \subseteq \pi_{empno}(EMP)$, the insertion of a new department respects this constraint if we consider the following active rule:

```
on insert(n, d, m) into DEPT
if $m \notin \pi_{empno}(EMP)$
then call insert_EMP(m, d)
```

where insert_EMP is an interactive program asking for a name and a salary for the new manager, so that the corresponding tuple can be inserted in the relation EMP.

Another important feature of active rules is their ability to express *dynamic* dependencies. The particularity of dynamic dependencies is that they refer to more than one database state (as opposed to static dependencies that refer to only one database state). A typical dynamic dependency, in the context of the database of Fig. 1, is to state that salaries must never decrease, which corresponds to the following active rule:

**on** update_sal(*ne*, *new-sal*) in EMP
**if** $new\text{-}sal > \pi_{\text{sal}}(\sigma_{\text{empno=ne}}(\text{EMP}))$
**then** set $sal = new\text{-}sal$ where $empno = ne$

where update_sal is the update meant to assign the salary of the employee number *ne* to the value *new-sal* and where the set instruction actually performs the modification.

Although active rules are an elegant and powerful way to specify various dynamic aspects of databases, they raise important questions concerning their execution. Indeed, as the execution of an active rule fires other active rules in its action, the main problem is to decide how these rules are fired. Three main execution modes have been proposed so far in the literature: the immediate mode, the deferred mode, and the concurrent mode. According to the immediate mode, the rule is fired as soon as its event occurs while the condition is true (this is the first case of our previous example). According to the deferred mode, the actions are executed only after the last event occurs and the last condition is evaluated (this corresponds to the second case of our previous example). In the concurrent mode, no policy of action execution is considered, but a separate process is spawned for each action and is executed concurrently with other processes. It should be clear that executing the same active rules according to each of these modes generally gives different results and the choice of one mode over the others depends heavily on the application. This is why, in most prototypes implementing active rules, the choice of the execution mode is left to the user.

### Transaction Management

Contrary to what has been discussed before, the problem of transaction management concerns the physical level of DBMSs and not the conceptual level. Although transaction execution is independent of the conceptual model of databases being used (relational or not), this research area has been investigated in the context of relational databases. The problem is that, in a multiuser environment, several transactions may have to access the same data simultaneously, and then, in this case the execution of these transactions may leave the database inconsistent whereas each transaction executed alone leaves the database in a consistent state (an example of such a situation will be given shortly). Additionally, modifications of data performed by transactions must survive possible hardware or software failures.

To cope with these difficulties, the following two problems have to be considered: (1) the concurrency control problem (that is, how to provide synchronization mechanisms which allow for efficient and correct access of multiple transactions in a shared database) and (2) the recovery problem (that is, how to provide mechanisms that react to failures in an automated way). To achieve these goals, the most prominent computational model for transactions is known as the *read-write* model, which considers transactions as sequences of read and write operations operating on the tuples of the database. The operation read($t$) indicates that $t$ is retrieved from the secondary memory and entered in the main memory, whereas the operation write($t$) does the opposite: the current value of $t$ in the main memory is saved in the secondary memory, and thus survives execution of the transaction. Moreover, two additional operations are considered, modeling, respectively, successful or failed executions: the *commit* operation (which indicates that changes in data must be preserved), and the *abort* operation (which indicates that changes in data performed by the transaction must be undone, so that the aborted transaction is simply ignored). For example, call the first tuple of the relationship EMP of Fig. 1, and assume that two transactions $T_1$ and $T_2$ increase John's salary of 500 and 1,000, respectively. In the read-write model, both $T_1$ and $T_2$ have the form: read($t$) ; write($t'$) ; commit, where $t'.sal = t.sal + 500$ for $T_1$ and where $t'.sal = t.sal + 1{,}000$ for $T_2$.

Based on these operations, many criteria for correctness of transaction execution have been proposed. We shall restrict ourselves to the most common, known as serializability of schedules. A schedule is a sequence of interleaved operations originating from various transactions, and a schedule built up from transactions $T_1, T_2, \ldots, T_k$ is said to be serializable if its execution leaves the database in the same state as the sequential execution of transactions $T_i$'s, in some order would do. In the previous example, let us consider the following schedule:

$$\text{read}_1(t)\ ;\ \text{read}_2(t)\ ;\ \text{write}_1(t_1)\ ;\ \text{commit}_1\ ;\ \text{write}_2(t_2)\ ;\ \text{commit}_2$$

where the subscripts correspond to the transaction where the instructions occur. This schedule is not serializable, because in execution corresponds neither to $T_1$ followed by $T_2$ nor to $T_2$ followed by $T_1$. Indeed, transactions $T_1$ and $T_2$ both read the initial value of $t$ and the effects of $T_1$ on tuple $t$ are lost, as $T_2$ commits its changes after $T_1$.

To characterize serializable schedules, one can design execution protocols. Here again many techniques have been introduced, and we focus on the most frequent of them in actual systems, known as the *two-phase locking protocol.* The system associates every read or write operation on the same object to a lock, respectively, a read-lock or a write-lock, and once a lock is granted for a transaction, other transactions cannot access the corresponding object. Additionally, no lock can be granted to a transaction that has already released a lock. It is easy to see that, in the previous example, such a protocol prevents the execution of the schedule we considered, because $T_2$ cannot read $t$ unless $T_1$ has released its write-lock.

Although efficient and easy to implement, this protocol has its shortcomings. For example, it is not free of deadlocks, that is, the execution may never terminate because two transactions are waiting for the same lock at the same time. For instance, transaction $T_1$ may ask for a lock on object $o_1$, currently owned by transaction $T_2$ which in turn asks for a lock on object $o_2$, currently owned by transaction $T_1$. In such a situation, the only way to restart execution is to abort one of the two transactions. Detecting deadlocks is performed by the detection of cycles in a graph whose nodes are the transactions in the schedule and in which an edge from transaction T to transaction T′ means that T is waiting for a lock owned by T′.

## RELATIONAL DATABASE SYSTEMS AND SQL

In this section, we describe the general architecture of relational DBMS, and we give an overview of the language SQL which has become a reference for relational systems.

### The Architecture of Relational Systems

According to a proposal by the ANSI/SPARC normalization group in 1975, every database system is structured in three main levels:

1. the *internal (or physical) level* which is concerned with the actual storage of data and by the management of transactions;
2. the *conceptual level* which allows describing a given application in terms of the DBMS used, that is, in terms of relations in the case of a relational DBMS; and
3. the *external level* which is in charge of taking user's requirements into account.

Based on this three-level general architecture, all relational DBMSs are structured according to the same general schema that is seen as two interfaces, the external interface and the storage interface.

The external interface, which is in charge of the communication between user's programs and the database, contains five main modules: (1) precompilers allowing for the use of SQL statements in programs written in procedural languages such as C, PASCAL, or COBOL; (2) an interactive interface for a real-time use of databases; (3) an analyzer which is in charge of the treatment of SQL statements issued either from a user's program or directly by a user via the interactive interface; (4) an optimizer based on the techniques discussed previously; and (5) a catalog, where information about users and about all databases that can be used, is stored. It is important to note that this catalog, which is a basic component for the management of databases, is itself organized as a relational database, usually called the metadatabase, or data dictionary.

The storage interface, which is in charge of the communications between database and the file management system, also contains five main modules: (1) a journal, where all transactions on the database are stored so that the system restarts safely in case of failures; (2) the transaction manager which generally works under the two-phase locking protocol discussed previously; (3) the index manager (indexes are created to speed up the access to data); (4) the space disk manager which is charge of defining the actual location of data on disks; and (5) the buffer manager which is in charge of transferring data between the main memory and the disk. The efficiency of this last module is crucial in practice because accesses on disks are very long operations that must be optimized. It is important to note that this general architecture is the basis for organizing relational system that also integrate network and distributed aspects in a client-server configuration or distributed database systems.

### An Overview of SQL

There have been many languages proposed to implement relational calculus. For instance, the language QBE (Query By Example) is based on domain calculus, where the language QUEL (implemented in the system INGRES) is based on tuple calculus. These languages are described in Ref. 2. We focus here on language SQL which is now implemented in all relational systems.

SQL is based on domain calculus but also refers to the tuple calculus in some of its aspects. The basic structure of a SQL query expression is the following:

```
SELECT ⟨list of attributes⟩
FROM ⟨list of relations⟩
WHERE ⟨condition⟩
```

which roughly corresponds to a relational expression containing projections, selections, and joins. For example, in the database of Fig. 1, the query **E** is expressed in SQL as follows:

```
SELECT EMP.deptno, dname
FROM EMP, DEPT
WHERE sal < 10,000 AND EMP.deptno = DEPT.deptno
```

We draw attention to the fact that the condition part reflects not only the selection condition from **E,** but also that, to join tuples from the relationships EMP and DEPT, their deptno values must be equal. This last equality must be explicit in SQL, whereas, in relational algebra, it is a consequence of the definition of the join operator. We also note that terms such as EMP.deptno or DEPT.deptno can be seen as terms from tuple calculus, whereas terms such as deptno or dname, refer to domain calculus. In general, prefixing an attribute name to the corresponding relationship name is required if this attribute occurs in more than one relationship in the FROM part of the query.

The algebraic renaming operator is implemented in SQL, but concerns relations, rather than attributes as in relational algebra. For example, the algebraic expression $\mathbf{E}_1$ (which computes the number of employees working in at least two distinct departments) is written in SQL as follows:

```
SELECT EMP.empno
FROM EMP,EMP EMPLOYEES
WHERE EMP.deptno⟨⟩EMPLOYEES.deptno AND
EMP.empno = EMPLOYEES.empno
```

Set theoretic operator's union, intersection, and difference are expressed as such in SQL, by the keywords UNION, INTERSECT, and MINUS (or EXCEPT), respectively. Thus, it turns out that every expression of relational algebra can be written as a SQL statement, and this basic result is known as the *completeness* of the language SQL. An important point in this respect is that SQL expresses more queries than relational algebra as a consequence of introducing functions (whereas function symbols are not considered in relational calculus) and "grouping" instructions in SQL. First, since relations are restricted to the first normal form, it is impossible to consider structured attributes, such as dates or strings. SQL overcomes this problem by providing usual functions for manipulating dates or strings, and additionally, arithmetic functions for counting or for computing minimum, maximum, average, and sum are available in SQL. Moreover, SQL offers the possibility of grouping tuples of relations, through the GROUP

BY instruction. As an example of these features, the numbers of departments together with the associated numbers of employees are obtained in the database of Fig. 1 with the following SQL query (in which no WHERE statement occurs, because no selection has to be performed):

```
SELECT deptno, COUNT(empno)
FROM EMP
GROUP BY deptno
```

On the other hand, a database system must incorporate many other basic features concerning the physical storage of tuples, constraints, updates, transactions, and confidentiality. In SQL, relations are created with the CREATE TABLE instruction, where the name of the relation together with the names and types of the attributes are specified. It is important to note that this instruction allows specifying constraints and information about the physical storage of the tuples. Moreover, other physical aspects are taken into account in SQL by creating indexes or clusters to speed up data retrieval.

Update instructions in SQL are either insertion, deletion, or modification instructions in which WHERE statements are incorporated to specify which tuples are affected by the update. For example, in the database of Fig. 1, increasing the salaries of 10% of all employees working in department number 1 is achieved as follows:

```
UPDATE EMP
SET sal = sal * 1.1
WHERE deptno = 1
```

Transactions are managed in SQL by the two-phase locking protocol, using different kinds of locks, allowing only read data or allowing read and write data. Moreover, activeness in databases is taken into account in SQL through the notion of *triggers,* which are executed according to the immediate mode.

Data confidentiality is a very important issue, closely related to data security, but has received very little attention at the theoretical level. Nevertheless, this problem is addressed in SQL in two different ways: (1) by restricting the access to data to specified users and (2) by allowing users to query only the part of the database they have permission to query. Restricting access to data by other users is achieved through the GRANT instruction, that is specified by the owner either on a relation or on attributes of a relation. A GRANT instruction may concern queries and/or updates, so that, for example, a user is allowed to query for salaries of employees, while forbidding the user to modify them. On the other hand, a different way to ensure data confidentiality consists in defining derived relations called *views.* For instance, to prevent users from seeing the salaries of employees, one can define a view from the relation EMP of Fig. 1 defined as the projection of this relation over attributes *empno, ename,* and *deptno.* A view is a query, whose SQL code is stored in the metadatabase, but whose result is not stored in the database. The concept of views is a very efficient tool for data confidentiality, thanks to the high expressional power of queries in SQL. However, the difficulty with views is that they are not updatable, except in very restricted cases. Indeed, because views are derived relations, updates on views must be translated into updates on the relations of the database, and this translation, when it exists, is generally not unique. This problem, known as the nondeterminism of view updating, is the subject of much research but has not yet been satisfactorily solved.

Now we conclude by mentioning that relational systems are successful in providing powerful database systems for many applications, essentially for business applications. However, these systems are not adapted to many new applications, such as geographical information systems, knowledge-based management, or data warehousing because of two kinds of limitations on the relational model:

1. Relations are flat structures which prevent easily managing data requiring sophisticated structures. This remark led to the emergence of object-oriented database systems that are currently the subject of important research efforts, most of them originating from concepts of object-oriented languages, and also from concepts of relational databases. As another research direction in this area, we mention the emergence of object-relational data models that extend the relational model by providing a richer type system including object orientation, and that add constructs to relational languages (such as SQL) to deal with the added data types. We refer to Ref. 8 for an introductory discussion on object-oriented databases and object-relational data models.
2. Relational algebra does not allow for recursiveness [see (3)], and thus, queries, such as the computing the transitive closure of a graph cannot be expressed. This remark has stimulated research in the field of deductive databases, a topic closely related to logic programming but which also integrates techniques and concepts from relational databases. The basic concepts of deductive databases and their connections with relational databases are presented in Ref. 2 and studied in full detail in Ref. 9.

Moreover, the general problem of dynamic aspects of databases is currently the subject of much research effort. This problem, known as the view updating problem in the field of relational databases, is also addressed in the object-oriented or deductive frameworks. The basic questions under investigation that have not yet been answered are What should be meant be updating a database and How can deduced data be updated? Furthermore, an important field of investigation, which is currently of much interest in the database community, is that of *data mining.* Indeed, it is becoming more and more crucial to extract abstracted information from many huge available databases, and in all these investigations, the relational model is the basic database model under consideration. Data mining and other emerging topics, such as data warehousing, are discussed in Ref. 8.

## BIBLIOGRAPHY

1. E. F. Codd, A relational model of data for large shared data banks, *Communication of the ACM,* **13**: 377–387, 1970.
2. J. D. Ullman, *Principles of Database and Knowledge-Base Systems,* Rockville, MD: Computer Science Press, 1988, Vol. I–II.
3. D. Maier, *The Theory of Relational Databases,* Rockville, MD: Computer Science Press, 1983.

4. P. C. Kanellakis, Elements of relational database theory, in J. Van Leuwen (ed.), *Handbook of Theoretical Computer Science,* Vol. B: *Formal and Semantics.* Amsterdam: North Holland, 1990, pp. 1073–1156.

5. J. W. Lloyd, *Foundations of Logic Programming,* 2nd ed., Berlin, Germany: Springer-Verlag, 1987.

6. E. F. Codd, Relational Completeness of Data Base Sublanguages, in R. Rustin (ed.), *Data Base Systems,* Englewood Cliffs, NJ: Prentice-Hall, 1972, pp. 65–98.

7. W. W. Armstrong, Dependency structures of database relationships. *Proc. IFIP Congress,* Amsterdam: North Holland, 1974, pp. 580–583.

8. A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts,* 3rd ed., New York: McGraw-Hill series in Computer Science, 1996.

9. S. Ceri, G. Gottlob, and L. Tanca, *Logic Programming and Databases,* Berlin: Springer-Verlag, 1990.

## *Reading List*

S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases,* Reading, MA: Addison-Wesley, 1995.

P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Reading: MA: Addison–Wesley, 1987. A good introduction and a fine reference source for the topic of transaction management.

R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems,* 2nd ed., Redwood City, CA: Benjamin Cummings, 1994. One of the most widely used database textbooks.

C. H. Papadimitriou, *The Theory of Database Concurrency Control,* Rockville, MD: Computer Science Press, 1986. A reference source for the theoretical foundations of concurrency control.

J. D. Ullman and J. Widom, *A First Course in Database Systems,* Englewood Cliffs, NJ: Prentice-Hall, 1997. A good and up-to-date textbook on databases.

M. Y. Vardi, Fundamentals of dependency theory, in E. Borger (ed.), *Trends in Theoretical Computer Science,* Rockville, MD: Computer Science Press, 1987, pp. 171–224. A complete introduction to theoretical aspects of dependency theory.

G. Vossen, *Data Models, Database Languages, and Database Management Systems,* Workingham, UK: Addison–Wesley, 1991. This book is a fine introduction to the theory of databases.

Dominique Laurent
University of Tours