On Multisets in Database Systems

Gianfranco Lamperti, Michele Melchiori, and Marina Zanella

Dipartimento di Elettronica per l'Automazione
Università di Brescia, Via Branze 38, 25123 Brecia, Italy
(lamperti|melchior|zanella)@ing.unibs.it

Abstract. Database systems cope with the management of large groups of persistent data in a shared, reliable, effective, and efficient way. Within a database, a multiset (or bag) is a collection of elements of the same type that may contain duplicates. There exists a tight coupling between databases and multisets. First, a large variety of data models explicitly support multiset constructors. Second, commercial relational database systems, even if founded on a formal data model which is set-oriented in nature, allows for the multiset-oriented manipulation of tables. Third, multiset processing in databases may be dictated by efficiency reasons, as the cost of duplicate removal may turn out to be prohibitive. Finally, even in a pure set-oriented conceptual framework, multiset processing may turn out to be appropriate for optimization of query evaluation. The mismatch between the relational model and standardized relational query languages has led researchers to provide a foundation to the manipulation of multisets. Other research has focused on extending the relational model by relaxing the first normal form assumption, giving rise to the notion of a nested relation and to a corresponding nested relational algebra. These two research streams have been integrated within the concept of a complex relation, where different types of constructors other than relation coexist, such as multiset and list. Several other database research areas cope with multiset processing, including view maintenance, data warehousing, and web information discovery.

1 Introduction

There exists a tight coupling between databases and multisets. A database is a collection of persistent information that is managed by a software package called *database management system* (DBMS) or, simply, *database system*. A database may be very large in nature. Consequently, a DBMS is required to manage information not only in main memory but, more specifically, in mass memory. Usually, a database is *shared*, that is, different applications and users are permitted to access the same database concurrently. This way, both redundancy and inconsistency of data can be avoided. A DBMS provides several other capabilities, including support for *recovery* and *privacy* of data.

A database is defined according to a *data model*, which provides means of data organization based on certain design patterns. A data model supports *structuring constructs* that are similar to the type constructors of general-purpose programming languages, like record, array, set, and file.

C.S. Calude et al. (Eds.): Multiset Processing, LNCS 2235, pp. 147–215, 2001.

Throughout the history of database technology, several data models have been defined, including the *hierarchical data model*, the *network data model*, the *relational data model*, and the *object-oriented data model*. All of them are called *logical* data models, to highlight that the relevant data structures reflect a specific organization, such as a tree, a graph, a table, or an object. Usually, a database is first described based on a *conceptual* data model, which is independent of the specific organization of data, such as the *entity-relationship data model*. Such a conceptual model is then mapped to a logical data model.

A database consists of a *schema* and an *instance*. The database schema defines the structure of data, which is expected to be essentially invariant with time. The database instance is the actual value of data, which is instead time-varying. The database changes its *state* when data are modified either by deleting or updating existing information, or by inserting new information.

The content of a database can be defined, accessed, and manipulated by means of a *database language*, which usually provides both *data definition* and *data manipulation* constructs. By 'data definition' we mean specifying the database schema. By 'data manipulation' we mean either querying or changing the database instance. For the DBMS to be usable, it is expected to retrieve and modify data efficiently, whatever the data model.

To simplify user interaction with the system, the same information can be organized at different *abstraction levels*. Specifically, the *physical level* describes how the data are stored in persistent memory in terms of complex low-level data structures. The *logical level* describes what data are incorporated within the database and what relationships exist among such data. Finally, the *view level* describes what part of the database can be seen by each class of users, as several views can be provided for the same database.

For the different abstraction levels to work as whole, appropriate mappings from the view level to the logical level, and from the logical level to the physical level, are required. Considering a relational DBMS, a view is mapped to one or several relations (tables) of the logical level, while a relation is mapped to one or several low-level data structures of the physical level[1].

Drawing now our attention to why multisets and databases are related to each other, we observe that, first, several DBMS data models support multiset constructors [15, 47, 28, 45, 36, 44, 23, 12]. Consequently, database query languages are affected by the use of duplicates.

Second, there exist practical multiset-oriented query languages that have been implemented based on previous formalization of some set-oriented languages. Considering relational database technology, relational DBMSs have been developed upon the mathematical notion of a relation and the formalization of set-oriented query languages, namely *relational algebra* and *relational calculus*. However, relational DBMSs, almost invariably extend the nature of relations to multisets and support query languages with additional expressive power.

[1] The physical level incorporates in general a variety of additional data structures that are invisible at the logical level, such as, for example, the index of a relation, which allows for fast query evaluation.

Finally, dealing with multisets rather than with sets is generally bound to speed up query evaluation, as there is no need for duplicate elimination, an operation which can be extremely costly in large databases. Note that, in this case, multiset processing may be confined to optimization issues within the query evaluation, even in set-oriented queries, where duplicates are removed only when the set-oriented semantics of a query would be otherwise violated.

The mismatch between the relational model and the standardized relational query languages has led researchers to provide a foundation to the manipulation of multisets. Other research has focused on extending the relational model by relaxing the first normal form assumption (which requires flat tables), giving rise to the notion of a nested relation and to a corresponding nested relational algebra. These two research streams have been integrated within the concept of a complex relation, which is polymorphic in nature, as different types of constructors other than relation coexist, such as multiset and list.

Several other database research areas cope with multiset processing, including view maintenance, data warehousing, and web information discovery.

The remainder of the paper is organized as follows. Section 2 introduces the relational model in a pure set-oriented framework. Section 3 discusses how multisets can be manipulated in relational database systems. Section 4 presents the nested relational model, where sets can be nested within each other to form nested relations. Section 5 provides some hints on how the nested relational model can cope with multisets, thereby yielding the notion of a complex relation. A discussion on related work is provided in Sect. 6. Conclusions are drawn in Sect 7.

2 Relational Model

Most database systems currently on the market are based on the relational paradigm, which was proposed by Codd in 1970 [16], with the main purpose of overcoming the typical flaws of the database systems of that period, among which was the inability to provide data independence, that is, the separation between the logical view of the model from its physical implementation.

However, the success of the relational approach was quite slow. Its high level of abstraction prevented for several years the implementation of efficient structures, as they were significantly different from those used in that epoch, typically, relevant to hierarchical and reticular data models. In fact, even though the first prototypes of relational systems have been developed since the early 1970s, the first relational systems appeared on the market in 1981, and became significant only in the middle 1980s.

The relational data model makes use of a single structure to organize data: the mathematical concept of n-ary relation. Beside being easy to formalize, the relational model has a simple representation: a relation is a table, and the database is viewed as a collection of tables. Another advantage of the relational model with respect to previous data models is that it responds to the requirement of data independence.

Intuitively, a table, which has a unique name in the database, contains one or more columns: each column has a heading, called attribute name, and a corresponding set of possible values, called the domain (for example, integers, strings, etc.). Each row is an ordered n-tuple of values $\langle v_1, v_2, \ldots, v_n \rangle$, each belonging to the domain of the corresponding attribute: each v_i is in the domain of column $i \in [1\,..\,n]$, where n is the number of columns. A table is therefore an unordered collection of distinct tuples.

Formally, let $\mathbb{U}$ be the set of attribute names, $\mathbb{U} = \{A_1, \ldots, A_n\}$, and $\mathbb{D}$ the set of domains, $\mathbb{D} = \{D_1, \ldots, D_n\}$. Each domain D_i contains only atomic values. We assume the existence of a function $\mathrm{Dom} : \mathbb{U} \mapsto \mathbb{D}$, which associates the appropriate domain with each attribute name. Then, a *tuple* over a set of attributes $X \subseteq \mathbb{U}$ is a function t that associates with each attribute $A_i \in X$ a value of the domain $\mathrm{Dom}(A_i)$; this value is denoted by $t_{(A_i)}$. We shall also write $t_{(Y)}$ with $Y \subseteq X$ to denote the restriction of the function t to the attributes in Y.

A *relation schema* of a relational database has the form $R = (A_1, \ldots A_n)$, where R is the relation name and A_i are distinct attribute names. A *database schema*, denoted by Δ, is a set of relation schemas with distinct relation names.

A *relation instance* (or simply, a *relation*) defined on a relation schema $R = (A_1, \ldots, A_n)$, is a finite set r of tuples over $X = \{A_1, \ldots, A_n\}$.

A *database instance* δ on a database schema Δ is a set of relations $\{r_1, \ldots, r_m\}$ where each r_i is defined on precisely one R_i in Δ.

With respect to the tabular representation of relation, the above definitions lead to the following observable properties:

1. The values within each column are homogeneous, and they all belong to the same domain. The domain corresponds to the attribute of the column.
2. There are no identical rows: a relation is a set, and therefore it does not contain duplicated elements.
3. The order of columns is irrelevant, since they are identified by their name.
4. The order of rows is immaterial, since they are identified by their content.

The relational data model is *value-oriented*, as all the information are represented by means of values: the identity of a tuple in a relation is based only on the values it determines for the attributes, and the relationship among tuples in different relations is based only on attribute values. This is particularly important since not all aspects of a relation might be known at a given time: in this case the relation could contain values which are not specified. The domains of attributes can be extended by including a special *null* value, that represents the absence of information. The theory of null values is especially important for the relational model, where all the information is value-based.

Moreover, part of the content of a relation is inherent to constraints on the values that some or all of its attributes can assume. Two important classes of such constraints are expressed by the notions of a *key* and *functional dependency*.

A subset K of the attributes of a relation r is a *key* of r when the following properties hold:

Table 1. Relation `Courses(course, year, teacher)`

course	year	teacher
Algebra	1	Sheila
Calculus	1	Gregory
Compilers	5	Jerri
Computers	2	Patricia
Databases	3	Carol
Geometry	2	Martin
Operating systems	4	Angelique
Programming languages	3	Jerri
Robotics	5	Richard
Software engineering	4	Angelique

1. *Unique identification*: r does not contain two distinct tuples t_1, t_2 which agree on all the attributes in K, that is:

$$(\forall t_1)(\forall t_2)(t_{1(K)} \neq t_{2(K)}) \ ; \tag{1}$$

2. *Minimality*: no proper subset of K enjoys the unique identification property.

A set K fulfilling the unique identification property is called a *superkey*, since it is the superset of a key. The set of all the attributes in a relation is always a key (since all the tuples are distinct), so each relation has at least one key.

Given two sets of attributes X and Y of a relation r, we say that Y *functionally depends* on X in r, denoted by $X \to Y$, if and only if for every pair of tuples $t_1 \in r$, $t_2 \in r$, if $t_{1(X)} = t_{2(X)}$, then $t_{1(Y)} = t_{2(Y)}$.

Intuitively, there is a functional dependency (FD) when the value of one or more attributes in a relation determines the value of another group of attributes. The concept of a key of a relation can be rephrased in terms of FD: a set K of attributes of r is a key if the following properties hold:

1. $K \to N_K$, where N_K is the set of attributes of r that do not belong to K;
2. No proper subset of K meets the same property.

Example 1. Shown in Tables 1 and 2 are relations `Courses` and `Teachers`, respectively. Specifically, the schema of `Courses` consists of three attributes, namely:

1. `course`, defined on the domain of course names, which identifies the course, in other words, `course` is a key for `Courses`;
2. `year`, defined on the range $[1\,..\,5]$, which identifies the curriculum year in which the course is taught;
3. `teacher`, defined on the domain of teacher names, which identifies the name of the relevant teacher.

The instance of `Courses` includes ten tuples. On the other hand, the schema of `Teachers` involves attributes `name`, `age`, and `department`, where `name` is a key.

Table 2. Relation `Teachers(name, age, department)`

name	age	department
Angelique	42	Computer science
Carol	38	Computer science
Gregory	63	Mathematics
Jerri	54	Computer science
Martin	42	Mathematics
Patricia	36	Electronics
Richard	45	Electronics
Sheila	40	Mathematics

Note that, in order to avoid dangling teachers, the domain of `name` in `Teachers` is expected to include the domain of `teacher` in `Courses`. Therefore attributes `teacher` and `name` allow us to join each tuple of `Courses` with one tuple of `Teachers` appropriately. Thus, to find out the department of the teacher of a given course, we match the appropriate tuple in `Courses` with the one in `Teachers` which agree on attributes `teacher` and `name`, respectively. □

2.1 Relational Algebra

Once a relational database schema has been instantiated, it is possible to retrieve and modify the stored information by means of appropriate languages. Formally, updating the database amounts to changing its state. Denoting with $\mathbb{S}$ the domain of states of a database schema Δ, the update language describes functions Φ between the domain of states, namely $\Phi : \mathbb{S} \mapsto \mathbb{S}$. On the other hand, querying a database amounts to creating new relations, therefore query languages describe functions from $\mathbb{S}$ to the set of relations over possible schemas. Relations yielded by queries are temporary, that is, they disappear automatically from the database after they have been displayed to the user as a side effect of the query evaluation. Thus, we distinguish *base relations* from *temporary relations*, where the former are those relations which instantiate the database schema, while the latter correspond to query results. However, temporary relations may be transformed into base (persistent) relations by special-purpose create statements.

A great deal of effort in the design of relational database systems has been devoted to query languages, among which is *relational algebra*. Relational algebra consists of expressions applied to relations. Intuitively, a relational-algebra expression follows the pattern of arithmetic expressions, where variables are replaced by relation names, while arithmetic operators, such as $+$, $-$, $*$, and $/$, are replaced by relational operators. The application of a relational operator to one or two relations yields a new relation, exactly as the application of an arithmetic operator gives rise to a new number. Likewise, relational operators may be applied to relational expression, exactly as arithmetic operators can be applied to arithmetic expressions, for example, $(x-y)*(z+w)$. Note that relational algebra is in fact an algebra because the application of relational operators on relations

Table 3. Relation `Priorities(course, prerequisite)`

`course`	`prerequisite`
Compilers	Software engineering
Computers	Calculus
Geometry	Algebra
Operating Systems	Programming languages
Programming languages	Computers
Robotics	Geometry
Robotics	Programming languages
Software engineering	Programming languages
Software engineering	Databases

produces relations. In other words, relational expressions are closed with respect to the domain of possible relations.

A *query* is an expression of relational algebra. Since relations are essentially sets, relational algebra deals with the manipulations of sets. However, it is possible to extend the semantics of relational operators to multisets, as outlined in Sect. 3. As a matter of fact, concrete relational query languages based somehow on relational algebra, such as SQL (see Sect. 3.4), allows the manipulation of relations in a multiset-oriented way, where, for efficiency reasons, duplicate tuples are not removed unless explicitly specified.

Relational-algebra operators include *projection*, *selection*, *product*, *join*, *renaming*, *union*, *intersection*, and *difference*. Furthermore, for modularity reasons, we introduce the *assignment* statement.

Projection. The projection operator is unary: it takes a relation and produces a new relation containing only a subset of its columns. Let r be a relation defined over the schema R containing the set X of attributes, and let $Y \subseteq X$. The *projection* of r onto Y, denoted by $\pi_Y(r)$, is a relation on the attributes in Y consisting of the restrictions of the tuples of r to the attributes in Y:

$$\pi_Y(r) \stackrel{\text{def}}{=} \{t_{(Y)} \mid t \in r\} \ . \tag{2}$$

Example 2. Shown in Table 3 is a relation called `Priorities`, whose schema includes attributes `course` and `prerequisite`. `Priorities` is meant to specify the prerequisites of each course, that is, the courses a student is required to have passed before taking the exam of that course. For example, `Geometry` requires `Algebra` as prerequisite. `Software engineering` requires two prerequisites, namely `Programming languages` and `Databases`. Note that both attributes `course` and `prerequisites` are part of the key. In fact, given a course, there exist in general several different prerequisites for it. On the other hand, a course may be a prerequisite for several different courses. As a matter of fact, `Programming languages` is a prerequisite for three courses. This means that neither `course` nor `prerequisite` is a key by its own, but only the composition of both.

Table 4. Result of relational-algebra Expression (3)

course
Software engineering
Calculus
Algebra
Programming languages
Computers
Geometry
Databases

To find out the courses which are prerequisites for some courses we write the following expression:

$$\pi_{\texttt{prerequisite}}(\texttt{Priorities}) \tag{3}$$

which will give the result outlined in Table 4. Note that, generally speaking, some of the tuples may become identical when they are projected on a set of attributes: in this case duplicated tuples are deleted, so the resulting relation may actually contain less tuples than the operand. In our example, two of the three tuples relevant to `Programming languages` have been eliminated from the result, so that the resulting relation includes seven tuples instead of nine. □

Selection. The selection operator is unary. Intuitively, the result of the selection is the subset of the tuples in the operand that satisfy a selection predicate expressed in terms of elementary comparisons of constants and attribute values plus logic connectives.

Propositional formula. Let r be a relation over the set of attributes X; a *propositional formula* $\wp$ over X is defined recursively as follows. *Atoms* over X have the form $A_1 \vartheta A_2$ or $A_1 \vartheta a$, where $A_1 \in X$, $A_2 \in X$, a is a constant, and ϑ is a comparison operator, $\vartheta \in \{=, <, >, \neq, \geq, \leq\}$. Every atom over X is a propositional formula over X; if $\wp_1, \wp_2$ are propositional formulas over X, then $\neg(\wp_1)$, $\wp_1 \wedge \wp_2$, and $\wp_1 \vee \wp_2$ are formulas over X. Parentheses can be used as usual. Nothing else is a formula. A propositional formula associates a Boolean value with each tuple in r.

Given a relation r over the schema R, the *selection* of r with respect to $\wp$, denoted by $\sigma_{\wp}(r)$, is a relation over the same schema R, containing the tuples of r that make $\wp$ true:

$$\sigma_{\wp}(r) \stackrel{\text{def}}{=} \{t \in r \mid \wp(t)\} \ . \tag{4}$$

Example 3. Considering Table 1, to find out the course names of the last two years we write:

$$\pi_{\texttt{course}}(\sigma_{\texttt{year}\geq 4}(\texttt{Courses})) \tag{5}$$

Table 5. Result of relational-algebra Expression (5)

course
Compilers
Operating Systems
Robotics
Software engineering

which is expected to produce the result outlined in Table 5. Note that the temporary result of the selection is the operand of the subsequent projection. □

Product. It is a binary operator. Let r_1 and r_2 be two relations defined over the set of attributes X and Y, respectively, such that $X \cap Y = \emptyset$. The *product* of r_1 and r_2, denoted by $r_1 \times r_2$, is a relation on $X \cup Y$ consisting of all the tuples resulting from the concatenation of tuples in r_1 with tuples in r_2:

$$r_1 \times r_2 \stackrel{\text{def}}{=} \{t \text{ over } X \cup Y \mid (\exists t_1)(\exists t_2)(t_1 \in r_1, t_2 \in r_2, t_{(X)} = t_1, t_{(Y)} = t_2)\} \ . \tag{6}$$

Join. It is a binary operator that comes in two versions, usually referred to as *natural join* and *theta-join*. Let r_1 and r_2 be two relations defined over the set of attributes XY and YZ, such that $XY \cap YZ = Y$. The *natural join* of r_1 and r_2, denoted by $r_1 \bowtie r_2$, is a relation on $XYZ = XY \cup YZ$ consisting of all the tuples resulting from the concatenation of tuples in r_1 with tuples in r_2 that have identical values for the attributes Y:

$$r_1 \bowtie r_2 \stackrel{\text{def}}{=} \{t \text{ over } XYZ \mid (\exists t_1)(\exists t_2)\ (t_1 \in r_1, t_2 \in r_2, t_{(XY)} = t_1, t_{(YZ)} = t_2\} \ . \tag{7}$$

Example 4. To associate with ech course and corresponding teacher the relevant prerequisite we write:

$$\pi_{course,teacher,prerequisite}(Courses \bowtie Priorities) \tag{8}$$

which generates the relation outlined in Table 6. □

Given two relations r_1 and r_2 over disjoint sets of attributes X and Y, a *theta-join* $r_1 \bowtie_{\wp} r_2$ is a relation over the set of attributes $X \cup Y$ containing tuples obtained by the concatenation of tuples of r_1 and r_2 that satisfy the propositional formula $\wp$, that is:

$$r_1 \bowtie_{\wp} r_2 \stackrel{\text{def}}{=} \{t \text{ over } X \cup Y \mid (\exists t_1)(\exists t_2)\ (t_1 \in r_1, t_2 \in r_2, t_{(X)}{=}t_1, t_{(Y)}{=}t_2, \wp(t))\} \ . \tag{9}$$

The theta-join can be expressed through the use of selection and product as follows:

$$r_1 \bowtie_{\wp} r_2 \ \equiv \ \sigma_{\wp}(r_1 \times r_2) \ . \tag{10}$$

Example 5. To retrieve, for each course, the department the corresponding teacher depends on we write:

$$\pi_{\texttt{course,department}}(\texttt{Courses} \bowtie_{\texttt{teacher=name}} \texttt{Teachers}) \tag{11}$$

which yields the relation outlined in Table 7. In this case we used a theta-join as the two linking attributes `teacher` and `name` have different identifiers. □

Renaming. It is a unary operator that only changes the name of the attributes in the result, leaving the content of the relation unchanged. It is used to overcome difficulties with those operators for which attribute names are significant. Let r be a relation defined over a set of attributes X, and Y another set of attributes with the same cardinality of X, that is, $|X| = |Y|$. Besides, let $A_1, A_2, \ldots, A_k$ and $A'_1, A'_2, \ldots, A'_k$ be an order for attributes in X and Y respectively. The *renaming*

$$\rho_{A'_1,\ldots,A'_k \leftarrow A_1,\ldots,A_k}(r) \tag{12}$$

is a relation which includes a tuple t' for each tuple $t \in r$, defined as follows: t' is a tuple on Y and $t'_{(A'_i)} = t_{(A_i)}$, $i \in [1\,..\,k]$. In practice, only the renamed attributes

Table 6. Result of relational-algebra Expression (8)

course	teacher	prerequisite
Compilers	Jerri	Software engineering
Computers	Patricia	Calculus
Geometry	Martin	Algebra
Operating systems	Angelique	Programming languages
Programming languages	Jerri	Computers
Robotics	Richard	Geometry
Robotics	Richard	Programming languages
Software engineering	Angelique	Databases

Table 7. Result of relational-algebra Expression (11)

course	department
Algebra	Mathematics
Calculus	Mathematics
Compilers	Computer science
Computers	Electronics
Databases	Computer science
Geometry	Mathematics
Operating systems	Computer science
Programming languages	Computer science
Robotics	Electronics
Software engineering	Computer science

will be indicated within the two lists $A_1, \ldots, A_k$ and $A'_1, \ldots, A'_k$, namely those for which $A_i \neq A'_i$, rather than the complete list of attributes.

Since the renaming operator is not intended to change the domain of the attributes, we also require $\mathrm{Dom}(A'_i) = \mathrm{Dom}(A_i)$.

Example 6. We may formulate the same operation of Example 5 by replacing the theta-join in Expression (11) with a renaming and a natural join as follows:

$$\pi_{\texttt{course,department}}(\texttt{Courses} \bowtie (\rho_{\texttt{teacher} \leftarrow \texttt{name}}(\texttt{Teachers}))) \tag{13}$$

which yields the same relation displayed in Table 7. □

It is worthwhile pointing out that a natural join can be expressed in terms of projection, theta-join, and renaming operations. Let r_1 and r_2 be two relations over schemas $\texttt{XY}$ and $\texttt{YZ}$, respectively, where $\texttt{XY} \cap \texttt{YZ} = Y$. Then, the following equivalence holds:

$$r_1 \bowtie r_2 \equiv \pi_{\texttt{XYZ}}(r_1 \bowtie_{\wp_{Y,Y'}} (\rho_{A'_1,\ldots,A'_k \leftarrow A_1,\ldots,A_k}(r_2)) \tag{14}$$

where $Y = \{A_1, \ldots, A_k\}$, $Y' = \{A'_1, \ldots, A'_k\}$, $\texttt{XY} \cap Y'Z = \emptyset$, and $\wp_{Y,Y'}$ is a predicate defined as follows:

$$\wp_{Y,Y'}(t) \stackrel{\text{def}}{=} \begin{cases} \mathit{true} & \text{if } t_{(Y)} = t_{(Y')} \\ \mathit{false} & \text{otherwise .} \end{cases} \tag{15}$$

Set-theoretic operators. Since relations are special classes of sets, the meaning of the well-known set-theoretic operators can be retained for relations as well. Specifically, the *union*, *intersection*, and *difference* of two relations r and s are denoted by $r \cup s$, $r \cap s$, and $r - s$, respectively. However, since the result is expected to be defined over an expected schema, these operators require r and s to share the same schema, otherwise the tuples of the resulting relation would refer to different sets of attributes.

Example 7. To find the courses which either depend on some prerequisite courses or are themselves prerequisites for other courses we may formulate the following expression:

$$(\pi_{\texttt{course}}(\texttt{Priorities})) \cup (\rho_{\texttt{course} \leftarrow \texttt{prerequisite}}(\pi_{\texttt{prerequisite}}(\texttt{Priorities}))) \tag{16}$$

whose result is displayed in Table 8. Incidentally, all the courses in `Courses` are incorporated in Table 8. Duplicate courses have been removed.

Instead. if we are to retrieve the courses which are both prerequisite for some courses and depend on other courses we have just to change the union operator in Formula (16) with an intersection as follows:

$$(\pi_{\texttt{course}}(\texttt{Priorities})) \cap (\rho_{\texttt{course} \leftarrow \texttt{prerequisite}}(\pi_{\texttt{prerequisite}}(\texttt{Priorities}))) \tag{17}$$

Table 8. Result of relational-algebra Expression (16)

course
Algebra
Calculus
Compilers
Computers
Databases
Geometry
Operating Systems
Programming languages
Robotics
Software engineering

Table 9. Result of relational-algebra Expression (17)

course
Computers
Geometry
Programming languages
Software engineering

whose result is outlined in Table 9.

Finally, the following expression yields the courses which require some other courses, but are not prerequisite of any course:

$$(\pi_{\texttt{course}}(\texttt{Priorities})) - (\rho_{\texttt{course}\leftarrow\texttt{prerequisite}}(\pi_{\texttt{prerequisite}}(\texttt{Priorities}))) \tag{18}$$

whose result is outlined in Table 10. □

Assignment. The assignment operator does not extend the expressive power of relational algebra. In other words, all the operations involving the assignment can be expressed as well using the operators defined above. However, the assignment operator is useful because it allows the specification of expressions in a more concise and modular fashion. An assignment has the following form:

$$w \leftarrow E(r_1, r_2, \ldots, r_n) \tag{19}$$

Table 10. Result of relational-algebra Expression (18)

course
Compilers
Operating Systems
Robotics

where w is an identifier and E a relational expression. In general, a query expressed in relational algebra consists of a (possibly empty) sequence of assignments and a final relational expression Q:

$$\begin{aligned} & w_1 \leftarrow E_1(r_1, r_2, \ldots, r_n) \\ & w_2 \leftarrow E_2(r_1, r_2, \ldots, r_n, w_1) \\ & \quad \ldots \\ & w_m \leftarrow E_m(r_1, r_2, \ldots, r_n, w_1, w_2, \ldots, w_{m-1}) \\ & Q(r_1, r_2, \ldots, r_n, w_1, w_2, \ldots, w_m) \ . \end{aligned} \tag{20}$$

Each assignment instantiates a temporary relation w_i whose schema and instance are determined by the corresponding expression E_i. However, relations w_i are only visible within Q and disappear as soon as the execution of Q terminates.

Example 8. Consider the problem of determining the age of the youngest teacher. Note that relational algebra does not provide any aggregation function like min to determine a single value (e.g. maximum, minimum, or average) starting from a set of values. Consequently we have to specify the relevant query by means of relational operators only. This can be done as follows:

$$\begin{aligned} & \texttt{Ages} \leftarrow \pi_{\texttt{age}}(\texttt{Teachers}) \\ & \texttt{NotMin} \leftarrow \pi_{\texttt{age}}(\sigma_{\texttt{age}>\texttt{age1}}(\texttt{Ages} \times (\rho_{\texttt{age1}\leftarrow\texttt{age}}(\texttt{Ages})))) \\ & \texttt{Ages} - \texttt{NotMin} \ . \end{aligned} \tag{21}$$

In the above sequence of expressions, the first two are assignments, while the third is the displayed result. `Ages` is a temporary relation incorporating the whole set of ages in `Teachers` (see Table 2). The other temporary relation `NotMin` represents the set of ages which cannot be the minimum value. This claim is supported by the evidence that each age is combined with each other age in `Ages` by means of the product (the renaming is necessary to avoid duplicated attribute names in the resulting schema). Then a selection is applied to the result of the product: only the ages which are greater than at least another age are selected. Consequently, `NotMin` will embody all the ages but the minimum, this being the age which is not greater than any other age. Hence, the final result is obtained by simply complementing `NotMin` with respect to `Ages` through the difference operation. The final result is therefore the singleton $\{36\}$, which incorporates the age of `Patricia`, the youngest teacher. □

3 Multisets in Relational Database Systems

The notion of a relation as a set of tuples is a simple and formal model of data. Being a set, a relation cannot include duplicates. Accordingly, all the relations involved in the examples of Sect. 2 do not incorporate any duplicate tuple. However, commercial relational database systems are almost invariably based on multisets instead of sets. In other words, tables are in general allowed to include duplicate tuples.

The main reason for considering relations as multisets in database systems is efficiency of query evaluation. Normally, base relations do not include duplicates. However, duplicate tuples can be generated within the evaluation of a relational-algebra expression which involves either projections or unions. For example, when we do a projection, if we want the result to be a set, we are required to compare each tuple in the result with all the other tuples in the result in order to be sure that tuples are not replicated. Generally speaking, duplicate removal is very expensive in time. Instead, if we are allowed to have a multiset as the result, then we merely project each tuple and add it to the result, regardless of other occurrences of the same tuple.

Likewise, when making the union of two relations r_1 and r_2, if we want the result to be a set, we must check that each tuple of one relation is not already included in the other. That is, if n_1 and n_2 are the number of tuples in r_1 and r_2, respectively, we are expected to make $n_1 \cdot n_2$ comparisons. Instead, if we accept a multiset as the result, then we just copy all the tuples of r_1 and r_2 into the answer, regardless of whether or not they appear in both relations, thereby making $n_1 + n_2$ operations without any comparison.

Sometimes, a hybrid approach is adopted: duplicate removals are delayed as much as possible in order to optimize efficiency whilst obtaining a set as the result. This means that, from the user point of view, relation are sets, while the evaluation system makes the best choices to produce the required result based on implementation-oriented criteria.

Example 9. With reference to Example 7, we may represent Expression (16) as a tree where leaves and internal nodes correspond to base relations and operators, respectively, as shown in Fig. 1.

A temporary relation is associated with each internal node, that implicitly represents the intermediate result of the expression inherent to the corresponding sub-tree.

Thus, the tree of Fig. 1 incorporates four internal nodes and two leaves, which incidentally happen to refer to the same base relation `Priorities`. Three nodes are virtually subjected to duplicate removals, these being the two projections

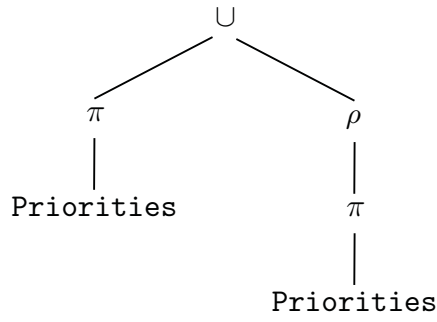

Fig. 1. Evaluation tree of Expression (16)

Table 11. Multiset result of Expression (16)

course
Compilers
Computers
Geometry
Operating Systems
Programming languages
Robotics
Robotics
Software engineering
Software engineering
Software engineering
Calculus
Algebra
Programming Languages
Computers
Geometry
Programming Languages
Programming Languages
Databases

and the union. If we allow the result to be a multiset, we will obtain the result displayed in Table 11.

Note that Table 11 incorporates 18 tuples while the set result of the same expression outlined in Table 8 includes 10 tuples only. However, the multiset result, although larger, can be computed more quickly. □

The instance of a multiset m can be implemented in two different ways:

1. *Extensional representation*: different occurrences of the same tuple are physically replicated, such as in Table 11.
2. *Intensional representation*: the whole collection of occurrences of the same tuple t is physically implemented by a single tuple (t, ω), where $\omega = \mathrm{Occ}(t, m)$.

Example 10. The intensional representation of the multiset displayed in Table 11 is shown in Table 12. □

Note that the intensional representation requires ω to be updated whenever duplicates may be generated, thereby losing the advantages of the extensional representation in terms of computation efficiency. On the other hand, representing intensionally several tuples by means of a single tuple allows the database system to save space and even time in those operations which do not introduce duplicates, as a smaller set of tuples are processed[2].

[2] From the physical point of view, a multiset stored intensionally is in fact a set.

Table 12. Intensional representation of the multiset displayed in Table 11.

`course`	ω
Compilers	1
Computers	2
Geometry	2
Operating Systems	1
Programming languages	4
Robotics	2
Software engineering	3
Calculus	1
Algebra	1
Databases	1

3.1 Relational Algebra for Multisets

The operators of relational algebra can be straightforwardly extended for manipulating multisets. However, as it will be shown in Sect. 3.3, some of the algebraic properties which hold for sets hold no longer for multisets. In what follows, multisets are enclosed within square brackets. Furthermore, the number of occurrences of t within a multiset m is denoted by the function $\mathrm{Occ}(t, m)$. For example, if m is the multiset displayed in Table 11 and $t = (\texttt{Software engineering})$, we have $\mathrm{Occ}(t, m) = 3$.

In order to univocally define a multiset, it is not enough to specify its elements, it is also necessary to assign a cardinality to each of them. Thus, in what follows, the definition of the multiset resulting from each operation is usually twofold. Indeed, the specification of the cardinality of each element is a complete definition as, if the cardinality of an element is zero, then such an element will not belong to the multiset. Therefore, in some cases, only the definition of the cardinality is given.

Projection of multisets. Provided that multiple occurrences of the same tuple can be inserted into a multiset, the definition of projection of multisets is formally the same given for sets. Let m be a multiset defined over the schema M containing the set X of attributes, and let $Y \subseteq X$. The projection of m onto Y, $\pi_Y(m)$, is a multiset on the attributes in Y consisting of the restrictions of the tuples of m to the attributes in Y:

$$\pi_Y(m) \stackrel{\text{def}}{=} [\, t_{(Y)} \mid t \in m \,] \ . \tag{22}$$

However, in contrast with Formula (2), duplicate restrictions of t on Y are not removed from the result. Specifically, the number of occurrences is given by the following formula:

$$\mathrm{Occ}(t, \pi_Y(m)) \stackrel{\text{def}}{=} \sum_{t' \in m, t'_{(Y)} = t} \mathrm{Occ}(t', m) \ . \tag{23}$$

Selection of multisets. The selection operation on a multiset m simply applies the selection predicate to each tuple in m and retains in the result those tuples $t \in m$ for which the predicate evaluates to true:

$$\sigma_{\wp}(m) \stackrel{\text{def}}{=} [\, t \in m \mid \wp(t) \,] \ . \tag{24}$$

Note that the Boolean value $\wp(t)$ is the same for all the occurrences of t in m. Consequently, depending on the value of $\wp(t)$, either all the occurrences of t are selected or none or them, that is:

$$\mathrm{Occ}(t, \sigma_{\wp}(m)) \stackrel{\text{def}}{=} \begin{cases} \mathrm{Occ}(t, m) & \text{if } \wp(t) \\ 0 & \text{otherwise} \ . \end{cases} \tag{25}$$

This is due to the inability to distinguish different occurrences of t within m based on $\wp$. Furthermore, as for sets, the selection operator does not generate new duplicates. However, it preserves the selected duplicates in the result.

Product of multisets. The product operation can be extended for multisets in a natural way. Let m_1 and m_2 be two multisets defined over schemas X and Y, respectively, such that $X \cap Y = \emptyset$. The product $m_1 \times m_2$ is a multiset over $X \cup Y$ consisting of all the tuples resulting from the concatenation of tuples in m_1 with tuples in m_2:

$$m_1 \times m_2 \stackrel{\text{def}}{=} [\, t \text{ over } X \cup Y \mid (\exists t_1)(\exists t_2)(t_1 \in m_1, t_2 \in m_2, t_{(X)} = t_1, t_{(Y)} = t_2) \,] \ . \tag{26}$$

In other words, each tuple of m_1 is paired with each tuple of m_2, regardless of whether it is a duplicate or not. As a result, if tuple t_1 appears n_1 times in m_1 and tuple t_2 appears n_2 times in m_2, then, denoting with $t_1 \oplus t_2$ the concatenation of tuples t_1 and t_2, in the product $m_1 \times m_2$, the tuple $t_1 \oplus t_2$ will appear $n_1 \cdot n_2$ times, that is:

$$\mathrm{Occ}(t_1 \oplus t_2, m_1 \times m_2) \stackrel{\text{def}}{=} \mathrm{Occ}(t_1, m_1) \cdot \mathrm{Occ}(t_2, m_2) \ . \tag{27}$$

Join of multisets. Both natural join and theta-join can be extended for multisets. Let m_1 and m_2 be two multisets defined over disjoint schemas X and Z, respectively. Then, the theta-join of m_1 and m_2 is defined as follows:

$$m_1 \bowtie_{\wp} m_2 \stackrel{\text{def}}{=} [\, t \text{ over } X \cup Z \mid (\exists t_1)(\exists t_2)\,(t_1 \in r_1, t_2 \in r_2, t_{(X)} = t_1, t_{(Y)} = t_2, \wp(t)) \,] \ . \tag{28}$$

The equivalence expressed in Equation (10) is required to hold for multisets as well, that is:

$$m_1 \bowtie_{\wp} m_2 \ \equiv \ \sigma_{\wp}(m_1 \times m_2) \ . \tag{29}$$

Therefore, based on the above equivalence, we can derive the number of occurrences of tuples in the theta-join result based on Equations (27) and (25) as follows:

$$\begin{aligned}\mathrm{Occ}(t_1 \oplus t_2, m_1 \bowtie_{\wp} m_2) &\stackrel{\text{def}}{=} \mathrm{Occ}(t_1 \oplus t_2, \sigma_{\wp}(m_1 \times m_2)) \\ &= \begin{cases} \mathrm{Occ}(t_1 \oplus t_2, m_1 \times m_2) & \text{if } \wp(t_1 \oplus t_2) \\ 0 & \text{otherwise} \end{cases} \\ &= \begin{cases} \mathrm{Occ}(t_1, m_1) \cdot \mathrm{Occ}(t_2, m_2) & \text{if } \wp(t_1 \oplus t_2) \\ 0 & \text{otherwise} \end{cases} . \end{aligned} \tag{30}$$

The natural join for multisets can be defined as follows. Let m_1 and m_2 be two multisets defined over the set of attributes $\mathtt{XY}$ and $\mathtt{YZ}$, respectively, where $\mathtt{XY} \cap \mathtt{YZ} = Y$. The natural join $m_1 \bowtie m_2$, is a multiset over $\mathtt{XYZ} = \mathtt{XY} \cup \mathtt{YZ}$ consisting of all the tuples resulting from the concatenation of tuples in m_1 with tuples in m_2 that have identical values for the attributes Y:

$$m_1 \bowtie m_2 \stackrel{\text{def}}{=} [\, t \text{ over } \mathtt{XYZ} \mid (\exists t_1)(\exists t_2)\ (t_1 \in m_1, t_2 \in m_2, t_{(\mathtt{XY})} = t_1, t_{(\mathtt{YZ})} = t_2\,] \ . \tag{31}$$

We require the multiset version of Equivalence (14) to hold, that is:

$$m_1 \bowtie m_2 \ \equiv\ \pi_{\mathtt{XYZ}}(m_1 \bowtie_{\wp_{Y,Y'}} (\rho_{A'_1,\ldots,A'_k \leftarrow A_1,\ldots,A_k}(m_2)) \tag{32}$$

where $\wp_{Y,Y'}$ is the predicate defined in Formula (15). Then, we make use of Formula (32) to compute the number of occurrences of tuples in the result. Let

$$m'_2 = \rho_{A'_1,\ldots,A'_k \leftarrow A_1,\ldots,A_k}(m_2) \ . \tag{33}$$

Then, based on Formulae (23) and (30), the number of occurrences of tuples in the natural join result can be derived as follows:

$$\mathrm{Occ}(t, m_1 \bowtie m_2) \stackrel{\text{def}}{=} \mathrm{Occ}(t, \pi_{\mathtt{XYZ}}(m_1 \bowtie_{\wp_{Y,Y'}} m'_2)) =$$

$$\sum_{t' \in (m_1 \bowtie_{\wp_{Y,Y'}} m'_2), t'_{(\mathtt{XYZ})} = t} \mathrm{Occ}(t', m_1 \bowtie_{\wp_{Y,Y'}} m'_2) =$$

$$\sum_{t' \in (m_1 \bowtie_{\wp_{Y,Y'}} m'_2), t'_{(\mathtt{XYZ})} = t, t'_{(\mathtt{XY})} = t'_1, t'_{(Y'Z)} = t'_2} \begin{cases} \mathrm{Occ}(t'_1, m_1) \cdot \mathrm{Occ}(t'_2, m'_2) & \text{if } \wp_{Y,Y'}(t') \\ 0 & \text{otherwise} \end{cases} =$$

$$\sum_{t'_{(\mathtt{XYZ})} = t, t'_{(\mathtt{XY})} = t'_1, t'_{(Y'Z)} = t'_2, t'_1 \in m_1, t'_2 \in m'_2, t'_{(Y)} = t'_{(Y')}} \mathrm{Occ}(t'_1, m_1) \cdot \mathrm{Occ}(t'_2, m'_2) \tag{34}$$

where $Y'Z = Y' \cup Z$.

Renaming of multisets. From the instance point of view, the renaming operation is an identity function. In other words, the result differs from the operand in the schema whilst retaining the instance. As such, it can be applied to multisets based on the same semantics defined for relations. In particular, given a multiset m, the number of occurrences of each tuple in the result does not change, that is:

$$\mathrm{Occ}(t, \rho_{A'_1, A'_2, \ldots, A'_k \leftarrow A_1, A_2, \ldots, A_k}(m)) \stackrel{\mathrm{def}}{=} \mathrm{Occ}(t, m) \ . \tag{35}$$

Set-theoretic operators for multisets. The set-theoretic operations defined for relations in Sect. 2.1, namely union, intersection, and difference, can be extended to multisets in a natural fashion. Let m_1 and m_2 be two multisets defined over the same schema. The union of m_1 and m_2 is defined as follows:

$$m_1 \cup m_2 \stackrel{\mathrm{def}}{=} [\, t \mid t \in m_1 \vee t \in m_2 \,] \ . \tag{36}$$

However, the peculiarity of the union definition lies in the fact that the resulting multiset is expected to incorporate all the occurrences of tuples, both in m_1 and m_2, that is:

$$\mathrm{Occ}(t, m_1 \cup m_2) \stackrel{\mathrm{def}}{=} \mathrm{Occ}(t, m_1) + \mathrm{Occ}(t, m_2) \ . \tag{37}$$

The intersection operation $m_1 \cap m_2$ will include a number of occurrences, of each shared tuple, equal to the minimum number of occurrences of the tuple in m_1 and m_2, respectively, that is:

$$\mathrm{Occ}(t, m_1 \cap m_2) \stackrel{\mathrm{def}}{=} \min(\mathrm{Occ}(t, m_1), \mathrm{Occ}(t, m_2)) \ . \tag{38}$$

In particular, if the tuple t is included in m_1 but not in m_2, it follows that, $\min(\mathrm{Occ}(t, m_1), \mathrm{Occ}(t, m_2)) = 0$, whatever $\mathrm{Occ}(t, m_1)$. That is, as expected, t will not be part of the intersection. On the other hand, if t is both included in m_1 and m_2, the result will include the smallest multiset of t, that is, the minimum number of occurrences of t in m_1 and m_2.

The difference operation $m_1 - m_2$ will include a number of occurrences of each tuple t which depends on whether or not $\mathrm{Occ}(t, m_1) \geq \mathrm{Occ}(t, m_2)$. Specifically, if this relationship is met, then the number of occurrences in the result will be the difference $\mathrm{Occ}(t, m_1) - \mathrm{Occ}(t, m_2)$, otherwise the result will not include t. Formally,

$$\mathrm{Occ}(t, m_1 - m_2) \stackrel{\mathrm{def}}{=} \max((\mathrm{Occ}(t, m_1) - \mathrm{Occ}(t, m_2)), 0) \ . \tag{39}$$

Duplicate removal. Given a multiset m, we may introduce a special unary operator called *duplicate removal*, denoted by $\delta(m)$, which generates a multiset obtained by removing from m duplicate occurrences of tuples. Formally, let

$$M \stackrel{\mathrm{def}}{=} \{T(t_1), T(t_2), \ldots, T(t_k)\} \tag{40}$$

be a partition of m where each part $T(t_i)$, $i \in [1\,..\,k]$, is the multiset of occurrences of tuple t_i. Then,

$$\delta(m) \stackrel{\text{def}}{=} [\, t_1, t_2, \ldots, t_k \,] \ . \tag{41}$$

Clearly,

$$\mathrm{Occ}(t, \delta(m)) \stackrel{\text{def}}{=} 1 \ . \tag{42}$$

3.2 Multiset Operations on Relations

Intuitively, each relation r can be viewed as a special multiset m in which each tuple happens to occur just once within the multiset, that is

$$(\forall t \in m)\ (\mathrm{Occ}(t, m) = 1) \ . \tag{43}$$

We say that m is the *multiset mirror* of r, denoted by $\mathfrak{M}(r)$. There exists an isomorphism between the tuples of r and m. More precisely, r and m incorporate the same tuples. Consequently, we may apply to r both the operators of relational algebra for relations and those extended for multisets, as defined in Sect. 3.1. This is due to the fact that there exists an isomorphism between the operators for relations and those for multisets, that is, for each operator of relational algebra there exists a homonymous operator of the extended algebra for multisets and vice versa. In other words, the algebraic operators are polymorphic in nature, as they can be applied either to relations or multisets, even though, generally speaking, with different semantics.

Let r, r_1, and r_2 denote relations. Let ω denote an (either unary or binary) generic (polymorphic) algebraic operator. Then, it is worthwhile finding out which operators meet the following equivalences:

$$\begin{array}{rcl} \mathfrak{M}(\omega(r)) & \equiv & \omega(\mathfrak{M}(r)) \\ \mathfrak{M}(r_1\ \omega\ r_2) & \equiv & \mathfrak{M}(r_1)\ \omega\ \mathfrak{M}(r_2) \end{array} \tag{44}$$

If Equivalence (44) holds for an operator ω, it means that considering the operand as either a relation or a multiset is the same, that is, it leads to the same result.

You can verify that all of the unary operators but projection meet the first equivalence in (44), that is:

$$(\forall \omega \in \{\sigma, \rho\})\ (\mathfrak{M}(\omega(r)) \ \equiv\ \omega(\mathfrak{M}(r))) \ . \tag{45}$$

The discrepancy of projection stems from the possible duplicate generation in multisets, which are instead removed in relations. Likewise, all of the binary operators but union meet the second equivalence in (44), that is:

$$(\forall \omega \in \{\times, \bowtie, \bowtie_{\wp}, \cap, -\})\ (\mathfrak{M}(r_1\ \omega\ r_2) \ \equiv\ \mathfrak{M}(r_1)\ \omega\ \mathfrak{M}(r_2)) \ . \tag{46}$$

To understand why union does not fulfill the equivalence, let us assume the case in which r_1 and r_2 contain the same tuple t. If so, the union of relations, $r_1 \cup r_2$, will contain just one instance of t, as the duplicate tuple is removed from the result. By contrast, the union of the corresponding multiset mirrors will embody two occurrences of t.

3.3 Algebraic Laws for Multisets

An algebraic law is an equivalence between two expressions of relational algebra. Each expression involves a number of variables denoting relations. The equivalence establishes that no matter what relations we replace for these variables, the two expressions give rise to the same result. However, an equivalence law which is valid for relations may happen not to hold when variables are interpreted as multisets.

A list of equivalence laws which are both valid for relations and multisets is given below, where x, y, and z denote collections (either relations or multisets) variables, A a subset of the attributes of the relevant projection operand, and $\wp_i$ a generic predicate on the tuples of the selection operand:

$$(x \cup y) \cup z \equiv x \cup (y \cup z) \tag{47}$$
$$(x \cap y) \cap z \equiv x \cap (y \cap z) \tag{48}$$
$$(x \bowtie y) \bowtie z \equiv x \bowtie (y \bowtie z) \tag{49}$$
$$x \cup y \equiv y \cup x \tag{50}$$
$$x \cap y \equiv y \cap x \tag{51}$$
$$x \bowtie y \equiv y \bowtie x \tag{52}$$
$$\pi_A(x \cup y) \equiv \pi_A(x) \cup \pi_A(y) \tag{53}$$
$$x \cup (y \cap z) \equiv (x \cup y) \cap (x \cup z) \tag{54}$$
$$\sigma_{\wp_1 \wedge \wp_2}(x) \equiv \sigma_{\wp_1}(x) \cap \sigma_{\wp_2}(x) \tag{55}$$

By contrast, the following algebraic laws hold for relations but not for multisets:

$$(x \cap y) - z \equiv x \cap (y - z) \tag{56}$$
$$x \cap (y \cup z) \equiv (x \cap y) \cup (x \cap z) \tag{57}$$
$$\sigma_{\wp_1 \vee \wp_2}(x) \equiv \sigma_{\wp_1}(x) \cup \sigma_{\wp_2}(x) \tag{58}$$

3.4 SQL and Multisets

Relational algebra provides a concise notation for defining queries. However, commercial database systems usually adopt a query language that is more user-friendly and even more powerful than relational algebra. The most commonly used concrete query language is SQL [13, 19, 37]. As a matter of fact, SQL has established itself as the standard relational-database language, which is not only a query language but also incorporates several other constructs, such as updating the database and defining security constraints.

Several different dialects (versions) of SQL exist. On the one hand, there are standards, among which are ANSI SQL and an updated standard called SQL2. There is also an emerging standard called SQL3 that extends SQL2 with several new capabilities, including recursion, triggers, and objects. On the other hand, there are many versions of SQL produced by the principal vendors of database

management systems, which usually conform completely to ANSI SQL, but also, to a large extent, to SQL2. To a lesser extent, some of them, include some of the advanced capabilities of SQL3.

The data model of SQL is relational in nature, as well as the relevant operations. However, unlike relational algebra, the tables manipulated by SQL are not relations, but, rather, multisets. The reason for this peculiarity is twofold. First, as mentioned in Sect. 2.1, this is due to a practical reason: since SQL tables may be very large, duplicate elimination might become a bottleneck for the computation of the query result. Second, SQL extends the set of query operators by means of aggregate functions, whose operands are in general required to be multisets of values, as illustrated in the sequel.

SQL query paradigm. In its simplest form, the typical SQL query is expressed by the so-called *select-from-where* paradigm, generically defined as follows:

$$\begin{array}{l} \texttt{SELECT } A_1, \ldots, A_k \\ \texttt{FROM } M_1, \ldots, M_n \\ \texttt{WHERE } \wp \end{array} \tag{59}$$

where each M_i, $i \in [1\,..\,n]$, is a table (multiset of tuples) name, each A_j, $j \in [1\,..\,k]$, is an attribute name, and $\wp$ a predicate. In terms of algebraic operators for multisets, the above query can be interpreted as follows:

$$\pi_{A_1,\ldots,A_k}(\sigma_{\wp}(M_1 \times \cdots \times M_n)) \ . \tag{60}$$

Therefore, the evaluation of the select-from-where statement is based on the following steps:

1. The tables listed in the `FROM` clause are combined through the product;
2. The corresponding result is filtered by the selection predicate $\wp$ specified in the `WHERE` clause;
3. The filtered tuples are projected on the list of attributes specified in the `SELECT` clause.

Example 11. With reference to the multiset mirrors of relations `Courses` and `Teachers` shown in Tables 1 and 2, respectively, we may display the name and the relevant department of those courses whose teachers are older than 40 as follows:

```
SELECT course, department
FROM Courses, Teachers
WHERE teacher = name AND age > 40
```

which yields the result outlined in Table 13. Incidentally, no duplicate tuples are generated. □

Table 13. Result of SQL query of Example 11

course	department
Calculus	Mathematics
Compilers	Computer science
Geometry	Mathematics
Operating systems	Computer science
Programming languages	Computer science
Robotics	Electronics
Software Engineering	Computer science

Example 12. With reference to the multiset mirrors of relations `Courses`, `Teachers`, and `Priorities`, displayed in Tables 1, 2, and 3, respectively, we may find out the courses which depend on some prerequisite course whose teacher is in the `Computer science` department as follows:

```
SELECT course
FROM Priorities, Courses, Teachers
WHERE prerequisite = course AND
      teacher = name AND department = 'Computer science'
```

which displays the result shown in Table 14. Note that, in contrast with the SQL query of Example 11, Table 14 is a multiset where the course `Software engineering` is replicated. □

Duplicate removal in SQL. In order for a result of a select-from-where statement not to include duplicate tuples, we are required to follow the keyword `SELECT` by the keyword `DISTINCT`, which tells SQL to produce only one occurrence of any tuple. Consequently, the result is guaranteed to be duplicate-free. In terms of the algebraic operators for multisets defined in Sect. 3.1, the set-oriented select-from-where paradigm

$$\begin{array}{l}\texttt{SELECT DISTINCT } A_1, \ldots, A_k \\ \texttt{FROM } M_1, \ldots, M_n \\ \texttt{WHERE } \wp\end{array} \tag{61}$$

Table 14. Result of SQL query of Example 12

course
Compilers
Operating Systems
Robotics
Software engineering
Software engineering

can be interpreted by prefixing Expression (60) with the duplicate removal operator as follows:

$$\delta(\pi_{A_1,\ldots,A_k}(\sigma_{\wp}(M_1 \times \cdots \times M_n))) \ . \tag{62}$$

In order to provide SQL queries with a set-oriented semantics, one might be tempted to place `DISTINCT` after every `SELECT`, on the assumption that it is harmless. On the contrary, it might be very expensive to perform duplicate removal on a table. Generally speaking, the table must be sorted so that the different occurrences of the same tuple appear next to each other. In fact, only by grouping the tuples in this way can we determine whether or not a given tuple should be eliminated. It is very likely that the time needed to sort the table is a good deal greater than the time it takes to execute the query itself. Consequently, the keyword `DISTINCT` should be used carefully.

Set-theoretic operations in SQL. SQL provides the usual set-theoretic operators, which can be applied to the results of queries, provided these queries produce tables with the same schema. Union, intersection, and difference operators are denoted in SQL by the keywords `UNION`, `INTERSECT`, and `EXCEPT`, respectively. However, unlike the `SELECT` statement, which preserves duplicates as a default and only eliminates them when instructed to by the `DISTINCT` keyword, the set-theoretic operators eliminate duplicates by default. We may change such a default by following `UNION`, `INTERSECT`, or `EXCEPT`, by the keyword `ALL`, thereby enabling the multiset semantics of such operators, as detailed in Sect. 3.1.

Example 13. We may rephrase in SQL Expression (16) of Example 7, which retrieves the courses which either depend on some prerequisite courses or are themselves prerequisites for other courses, as follows (the keyword `AS` allows us to change the name `prerequisite` into `course`, that is, to implement the renaming):

```
(SELECT course FROM Priorities)
    UNION
(SELECT prerequisite AS course FROM Priorities)
```

obtaining the same result displayed in Table 8. If, instead we write

```
(SELECT course FROM Priorities)
    UNION ALL
(SELECT prerequisite AS course FROM Priorities)
```

our result will preserve the duplicate tuples, as outlined in Table 15.

Compared with Table 8, the result in Table 13 includes duplicates for courses `Robotics`, `Software engineering`, `Programming languages`, `Computers`, and `Geometry`. However, apart from duplicates, Table 13 contains the same information of Table 8, even if with some redundancy. By contrast, it is worthwhile

Table 15. Result of multiset interpretation of Expression (16)

course
Compilers
Computers
Geometry
Operating Systems
Programming languages
Robotics
Robotics
Software engineering
Software engineering
Software engineering
Calculus
Algebra
Programming languages
Computers
Geometry
Programming languages
Programming languages
Databases

highlighting that the multiset counterpart of Expression (18) is inconsistent with the given query, that asks for the courses which require some other courses but are not prerequisite of any course:

```
(SELECT course FROM Priorities)
    EXCEPT ALL
(SELECT prerequisite AS course FROM Priorities)
```

Based on the multiset version of the difference operation defined in Sect. 3.1, the above SQL query gives rise to the result displayed in Table 16. Comparing Table 16 with Table 10, we note two discrepancies. First, in Table 16, `Robotics` is replicated, due to the projection within the first operand of the `EXCEPT` operator onto `course`. This is however only a redundancy. Second, Table 16 embodies the course `Software engineering`, which is instead not included in Table 10. Such a 'spurious' tuple stems from the semantics of difference operation for multisets,

Table 16. Result of multiset interpretation of Expression (18)

course
Compilers
Operating Systems
Robotics
Robotics
Software engineering

in which, according to Formula (39), the number of occurrences of each tuple t = "`Software engineering`" in the result is

$$\mathrm{Occ}(t, m_1 - m_2) = \max((\mathrm{Occ}(t, m_1) - \mathrm{Occ}(t, m_2)), 0) = \max((2-1), 0) = 1 \ . \tag{63}$$

However, we cannot obtain the required result by simply removing the `ALL` qualifier from `EXCEPT`, thereby writing the following SQL query:

```
(SELECT course FROM Priorities)
    EXCEPT
(SELECT prerequisite AS course FROM Priorities)
```

In fact, as duplicate removal is activated after the difference operation, the above statement will simply remove the duplicate course `Robotics`, preserving however the spurious tuple of `Software engineering`. To implement the correct query, we should also eliminate duplicates from the first operand as follows:

```
(SELECT DISTINCT course FROM Priorities)
    EXCEPT
(SELECT prerequisite AS course FROM Priorities)
```

which yields the expected result displayed in Table 10. □

Aggregate operations in SQL. Among other capabilities, SQL provides *aggregate functions*. An aggregate function takes a collection of values as input, which is in general a multiset, and returns a single value. Let a be the multiset of values of an attribute A of a table m, and $\mathcal{F}$ an aggregate function. Then,

$$\mathcal{F}(a) \tag{64}$$

is expected to return a scalar value based on a. SQL provides the following aggregate functions:

1. `SUM`, the sum of the values of a;
2. `AVG`, the average of values of a;
3. `MIN`, the least value in a;
4. `MAX`, the greatest value in a;
5. `COUNT`, the number of values in a.

Virtually, the computation of $\mathcal{F}(a)$ is performed by projecting a table m on an attribute A and by applying $\mathcal{F}$ to the resulting multiset a of A-values. Accordingly, from the syntactical point of view, in SQL the argument of the aggregate function is the identifier of an attribute A belonging to a table listed in the `FROM` clause.

Example 14. With reference to the multiset mirror of relation `Teachers` shown in Table 2, we may compute the average age of the teachers as follows:

```
SELECT AVG(age)
FROM Teachers
```

which gives rise to the value 45. Instead, if the average function would be applied to a set of values, we would obtain a result which is slightly greater than the actual average, owing to the elimination of one of the two occurrences of value 42, which is associated with both `Angelique` and `Martin`. □

Not all of the aggregate functions are sensitive to duplicate elimination of elements within the operand. Specifically, both the least and the greatest values within a collection a are preserved after duplication removal. We can express this property by means of the δ operator introduced in Sect. 3.1 as follows:

$$(\forall \mathcal{F} \in \{\texttt{MIN}, \texttt{MAX}\})\ (\mathcal{F}(a) \equiv \mathcal{F}(\delta(a)))\ . \tag{65}$$

Example 15. To determine the age of the youngest teacher in Table 2, we may rephrase the relational-algebra sequence of statements (21) of Example 8 by means of the following SQL query:

```
SELECT MIN(age)
FROM Teachers
```

which is expected to yield the value 36, relevant to `Patricia`. Note that the SQL version of the query is much simpler than its relational-algebra counterpart[3]. According to Equivalence (65), the same result would be obtained by prefixing `age` with the keyword `DISTINCT`. □

By contrast, the other SQL aggregate functions return in general different results, depending on whether duplicates are removed or not, that is:

$$(\forall \mathcal{F} \in \{\texttt{SUM}, \texttt{AVG}, \texttt{COUNT}\})\ (\mathcal{F}(a) \not\equiv \mathcal{F}(\delta(a)))\ . \tag{66}$$

To force duplicate removal, the attribute identifier must be preceded by the keyword `DISTINCT`.

Example 16. With reference to the multiset mirror of relation `Teachers` shown in Table 2, we may compute the number of departments as follows:

```
SELECT COUNT(DISTINCT department)
FROM Teachers
```

which produces the value 3. Note that the qualification `DISTINCT` is essential to tell SQL to discard duplicates before computing the number of departments. □

[3] Among the SQL aggregate functions, only `MIN` and `MAX` can be implemented in relational algebra. Thus, the expressive power of SQL is greater than the expressive power of relational algebra.

Table 17. Result of SQL query of Example 17

department	numc
Computer science	5
Mathematics	3
Electronics	2

SQL allows aggregate functions to be applied to parts of a table, which are collected based on a grouping criterion, expressed by means of the `GROUP BY` clause. More precisely, the multiset of tuples of a table m is partitioned based on a set G of attributes, thereby obtaining a partition of m where each part is composed of all the tuples which share the same values for attributes in G. The aggregate function will be applied to each of these parts and, as a result, a set of values, one for each part, will be computed.

Example 17. With reference to the multiset mirrors of relations `Courses` and `Teachers` shown in Table 1 and Table 2, respectively, we may compute the number of courses relevant to each department as follows:

```
SELECT department, COUNT(course) AS numc
FROM Courses, Teachers
WHERE teacher = name
GROUP BY department
```

which yields the result displayed in Table 17. The special clause `GROUP BY` tells SQL to make a partition of the join of `Courses` and `Teachers` before applying the `COUNT` aggregate function to each part, that is, to each set of tuples relevant to each department. Accordingly, the `GROUP BY` clause is applied after the `WHERE` clause and before `COUNT`. □

Even in the case of grouping, SQL aggregate functions are applied, by default, to multisets of values. The qualifier `DISTINCT` is used to force duplicate removal as usual.

Subqueries and test for absence of duplicates. SQL allows the `WHERE` clause to include SQL subqueries, whose result may represent the operand of a Boolean function included in the relevant predicate. For example, we may test for the membership of a value within a collection resulting from an SQL subquery by means of the `IN` Boolean function.

Example 18. Consider the multiset mirror of relation `priorities` in Table 3 and the problem of determining the courses which have some prerequisites but are not a prerequisite for any courses, which was already formulated in relational-algebra Expression (18). We may rephrase such a query in SQL either using the `EXCEPT` operator or by means of a selection in which the `WHERE` clause involves an SQL subquery as follows:

```
SELECT course
FROM Courses
WHERE course IN (SELECT course
                 FROM Priorities) AND
      course NOT IN (SELECT prerequisite
                     FROM priorities)
```

Each tuple in `Courses` is selected if and only if the relevant `course` appears within the set of courses in `Priorities` but not in the set of prerequisite courses. □

Example 19. To show how SQL allows us to test the absence of duplicates in a subquery, consider the multiset mirror of relation `Courses` displayed in Table 1. We may find out the courses who are taught by a teacher who teaches just that course as follows:

```
SELECT course
FROM Courses
WHERE UNIQUE (SELECT course
              FROM Courses AS C
              WHERE C.teacher = Courses.teacher
```

Note the use of the qualifier `AS` in the `FROM` clause, that renames `Courses` to `C` within the scope of the subquery. Besides, the pathname notation allows us to access tuples either in `Courses` or `C`. Thus, for each tuple of `Courses`, the predicate of the `WHERE` clause is evaluated by selecting on a copy of `Courses`, named `C`, the tuples matching the teacher name. Specifically, the result of each subquery returns the multiset of courses which are taught by the current teacher in the external query. Based on the `UNIQUE` Boolean function, the teacher is selected if and only if no duplicates are incorporated in the subquery, that is, if and only if the teacher teaches just one course, as required. Note that, if we substitute `UNIQUE` by `NOT UNIQUE`, we will retrieve the courses which are taught by a teacher who teaches several courses, that is, the complement of the previous query (the new predicate is in fact the logical negation of the previous one). □

Database modifications in SQL. Besides query-oriented language capabilities, SQL provides a means of changing the state of the database, specifically, by inserting tuples into a relation, by deleting certain tuples from a relation, or by updating the values of certain attributes in certain tuples.

Example 20. With reference to the multiset mirrors of relation `Priorities` shown in Table 3, we may insert the prerequisite `Algebra` for `Robotics` as follows:

```
INSERT INTO Priorities
VALUES ('Robotics', 'Algebra')
```

Conversely, we may remove all the prerequisites for `Robotics` as follows:

```
DELETE FROM Priorities
WHERE course = 'Robotics'
```

Finally, with reference to Table 1, we may move the courses of `Angelique` to one year upward as follows:

```
UPDATE Courses
SET year = year +1
WHERE teacher = 'Angelique'
```

□

There are some remarkable points about how insertions and deletions interact with the duplicates included in SQL tables. On the one hand, an insertion adds the specified tuple irrespective of whether it is already included in the table. On the other, the deletion statement, which appears to describe a single tuple to be deleted, in fact removes in general several tuples, because there is no way in SQL to remove one occurrence only from a collection of identical tuples. This characteristics is bound to surprising effects: given a table m containing a tuple t, if we followed the insertion of another occurrence of t into m by the deletion of t, both occurrences of t would be removed from m. In other words, the insertion of t followed by the removal of t would leave the database in a different state from what it was before the two operations.

4 Nested Relational Model

The success of relational database systems was mainly due to the fact that they provide a satisfactory response to the typical needs of business applications, for which the idea of databases as large collections of persistent data to be handled in an effective, efficient, and reliable way was conceived.

The most successful features of the relational model are the ease of use of its query language, which is set-oriented in nature, compared with the procedural, navigational style of earlier proposals, together with the conceptual simplicity of the data model. In fact, the relational model is actually based on a single data structure, the relation. Each relation contains its own data, and connections between data of different relations are implicitly represented by means of equality of values. For this reason, as already remarked, the relational model is often qualified as value-oriented.

Business applications usually involve large amounts of data with a relatively simple structure. The relational model provides an effective and implementation-independent way of specifying this structure while allowing at the same time flexible and sophisticated querying capabilities through set-oriented operations that act on whole relations rather than a single tuple at a time. For all these

reasons the relational model proved itself satisfactory with respect to the requirements of business applications, improving notably the productivity of software development in this area.

This success stimulated the adoption of the database technology in areas different from business applications, such as computer-aided design, computer-aided software engineering, knowledge representation, office systems, and multimedia systems. New applications, however, highlighted a number of shortcomings inherent to the relational database technology, among which are the following:

1. The involved data have a complex structure that cannot be expressed in a natural way in the relational model;
2. The relationships among data that derive from their semantics are very complex and cannot be efficiently stored in a value-oriented way;
3. Relational languages lack expressive power for most applications outside the business area.

The first step that was devised in the direction of widening the range of applicability of database systems was to extend the relational data model. This idea can quite naturally be understood starting from the consideration that two out of the above three limitations of relational systems arise from the simplicity of their data model. It looked reasonable to extend the data model, without losing its positive features, in order to explicitly represent data structures more complex than flat tuples of values. This would have also solved the second problem, that is, efficiently storing related data. As to the solution of the third problem, relational languages should have been extended in order to cope with more complex data structures, while retaining their set-oriented, declarative style: this extension, in the original idea, should have supplied the lacking expressive power.

Perhaps, the most noteworthy extension of the relational model was the *nested relational model*, according to which the assumption of atomic attributes (flat relations) is relaxed. Such an assumption, which excludes the possibility that an attribute value be a collection of other values, is called *First Normal Form* (1NF).

The standard relational model that derived implicitly from this assumption is therefore a *flat* relational model. Then, a relational database schema consists of relation schemas of the form:

$$R(A_1 : D_1, \ldots, A_n : D_n) \tag{67}$$

where each D_i is an atomic domain. The easiest way of defining a data model that allows for the representation of complex data structures is the direct extnsion of the relational model obtained by relaxing the 1NF assumption. A *nested relation* is defined in terms of (possibly complex) attributes. A *complex* attribute is in turn a (possibly nested) relation. Nested relations can be manipulated by means of special purpose languages [30, 6, 22, 1, 18, 8, 3, 43, 14, 10], among which are various extensions of relational algebra. In the *nested* data model, also called *Non First Normal Form* ($\neg$ 1NF), attribute values can be nested relations themselves, with unbounded depth. A nested relation schema R could be expressed as follows:

$$R(A_1 : T_1, \ldots, A_n : T_n) \tag{68}$$

where each T_j is either an atomic domain D or a nested relation schema of the form:

$$(A_{j1} : T_{j1}, \ldots, A_{jn_j} : T_{jn_j}) \ . \tag{69}$$

In the following, when irrelevant, atomic domains will be omitted.

As the notation suggests, the notion of a nested relation is the natural extension of the notion of a flat relation. A nested relation is a set of nested tuples in just the same way as a flat relation is a collection of flat tuples. A nested tuple associates a value from the corresponding domain with each attribute in the schema, as in the flat case. A flat tuple is a particular case of nested tuple, in which all the attributes are associated with an atomic domain. The value associated with an attribute is atomic if the attribute is simple, otherwise it is a nested relation. In the latter case the attribute is complex.

Even if the definition of nested relations is recursive, the schema is supposed to have a finite depth, because at each level every complex attribute must correspond to a new relation schema: it cannot be associated with a relation schema of the upper levels. Cyclical paths in the schemas are not allowed: the schema of a nested relation can therefore be conveniently represented by a tree, where the root is the name of the external nested relation, simple attributes are represented as leaves, and complex attributes correspond to internal nodes.

Nested relations and complex attributes have exactly the same properties as far as their structure is involved: we make use of the same definition for both of them. However, a nested relation is an instance consisting in a single set of nested tuples. By contrast, there are as many instances of complex attributes as tuples in its parent relation. Therefore, if the (external) nested relation instance consists of n tuples, a complex attribute corresponds to n sets of tuples. This asymmetry notably complicates the query language, even if it allows the expression of most queries more concisely.

Another consequence of this extension is that it gives a more complex semantics to elementary operations on tuples and attributes. For example, in the flat case, comparing two attribute values involves only a comparison of two atomic values, while in the nested model it requires a more complex comparison between sets when the attributes are complex.

Moreover, new operators are required for attribute comparison, such as in selection predicates. Besides the classical comparison operators, such as $=$, $>$, and $\geq$, it is necessary to introduce a number of relational operators for sets, such as $\supset$ (inclusion). Other set-oriented operations like $\cup$ (union) and $\cap$ (intersection) need also be included in the language, not only for relations, as in the flat case, but also for attributes.

Formally, an *extended relational database schema* S is a collection of rules. Each rule has the form

$$R = (R_1, \ldots, R_n) \tag{70}$$

where $R, R_1, \ldots, R_n$, which are called *names*, are distinct and there is no ordering on $R_1, \ldots, R_n$. The names on the right-hand side of the rule R form a set, which is denoted by E_{R}. Each rule has a different name on the left-hand side.

A name is a *higher-order* name if it occurs on the left-hand side of some rule; otherwise, it is *zero order*, or *attribute name*. Higher order names correspond to nested relations or attributes with a nested structure, while zero order names are ordinary, atomic attributes. Rules in the database schema associate each name with its structure. Since the structure is expected to have a finite depth, the structure of rules cannot be cyclic: this requirement will be specified below.

A name is *external* if it occurs only on the left-hand side of some rule; otherwise, it is *internal*. External names correspond to complex relations of the database, while internal names are attributes, either simple or complex.

Given an external name R in an extended relational database schema S, consider the smallest subset S' of S including:

1. The rule with R on the left-hand side;
2. For each higher-order name R_k on the right-hand side of some rule in S', the rule with R_k on the left-hand side.

S' is called the (nested) *relation schema* corresponding to R. The set S' corresponds to the rules in S that are accessible from R. We will normally identify a relation schema by its external name R rather than by listing explicitly the set S' of rules.

Given a relation schema R, we can define a unique *schema tree* of R, written G_{R}. Thus the schema tree of a relation schema S' with external name R is a tree rooted in R. The internal nodes of the tree will be the left-hand sides of other rules in S', and the leaves of the tree will be zero-order objects in the rules of S', representing basic attributes. Each non-leaf node of a relation schema tree represents a collection of tuples, each composed of the children of the node. The nodes of G_{R} are exactly the names in the rule R. G_{R} contains a directed edge from R to R' if and only if $R' \in E_{\mathrm{R}}$.

An extended relational database schema S consists of one or several relation schemas, possibly sharing some attributes. A unique *schema graph* of S, written G_{S}, can be defined, which is a graph resulting from the merging of the schema trees of all the relation schemas included in S.

An extended relational database schema S is *valid* if and only if G_{S} is a directed acyclic graph: this implies that the hierarchical structure of a nested relation has an unlimited but finite depth.

Two valid nested relation database schemas are *equivalent* if their schema graphs are isomorphic, in other words, if the graphs are equal up to renaming of internal (non-leaf) nodes.

Example 21. Considering the relations `Courses`, `Teachers`, and `Priorities` displayed in Tables 1, 2, and 3, respectively, we may define an extended database schema S as follows:

$$\begin{aligned} \texttt{Departments} &= (\texttt{department}, \texttt{teachers}) \\ \texttt{teachers} &= (\texttt{name}, \texttt{age}, \texttt{courses}) \\ \texttt{courses} &= (\texttt{course}, \texttt{year}) \\ \texttt{Priors} &= (\texttt{course}, \texttt{prerequisites}) \\ \texttt{prerequisites} &= (\texttt{prerequisite}) \end{aligned} \tag{71}$$

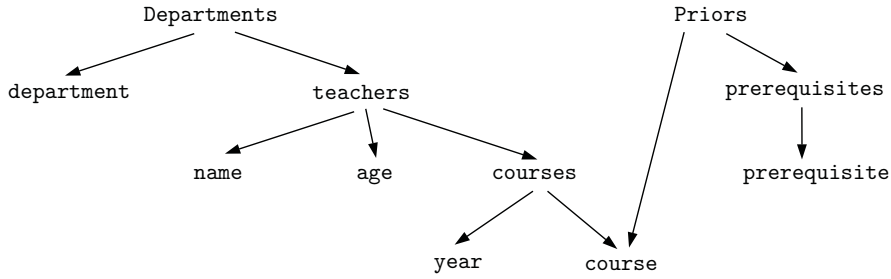

Fig. 2. Schema graph G_S of the extended database schema defined by Rules (71)

According to our definitions, this schema embodies two external names only, namely `Departments` and `Priors`. Higher-order names include `Departments`, `teachers`, `courses`, `Priors`, and `prerequisites`. Zero-order names are `name`, `age`, `course`, `department`, `year`, and `prerequisite`. The relevant schema graph G_S, which happens to be valid, is shown in Fig. 2. □

Having defined an extended relational database schema, we now turn to the problem of defining an instance of a complex relation. We want to define instances so that they are independent of the order of columns. We have simply to associate with each atomic attribute a single value taken from an appropriate domain, and to each higher-order name a set of tuples of the appropriate type. Since the ordering of names is immaterial, each value must be labeled by its name. Like for the flat relational model, we assume the existence of a function Dom that associates with each zero-order name its domain.

An instance of a name R, denoted by r, is an ordered pair $\langle R, V_R \rangle$, where V_R is a *value* for name R. If R is a zero order name, a value is an element of $\mathrm{Dom}(R)$. If R is a higher-order name, a value is a set $\{t\}$ of tuples t, where t contains a component $\langle R_i, V_{R_i} \rangle$ for each $R_i \in E_R$.

Note that two different complex relations can share part of the schema, that is, they can have some inner attribute with the same name and schema; however, they cannot share part of the instance: every relation is completely independent of any other relation. That is, nested tuples cannot share common sub-objects, and every update on a tuple or relation is confined to the tuple or relation.

The schema and instance associated with the same external name R form a *structure*. A structure is therefore a pair (R, r), where R is an external name and r an instance of R. A *database structure* (S, s) is a database schema together with an instance for its external names.

Example 22. With reference to Example 21, instances of the nested relations `Departments` and `Priors` are shown in Tables 18 and 19, respectively.

The instance of the nested relation `Departments` is composed of three tuples, each of which refers to a specific department. Associated with each department is a set of teachers, each of which is characterized by a name, an age, and a set of courses. A course is described by a name and the year in which it is taught.

Table 18. Instance of the nested relation `Departments`

department	teachers()			
	name	age	courses()	
			course	year
Computer science	Angelique	42	Operating systems	4
			Software engineering	4
	Carol	38	Databases	3
	Jerri	54	Compilers	5
			Programming languages	3
Electronics	Patricia	36	Computers	2
	Richard	45	Robotics	5
Mathematics	Gregory	63	Calculus	1
	Martin	42	Geometry	2
	Sheila	40	Algebra	1

As such, the instance of `Departments` incorporates in a hierarchical fashion all the information contained in Tables 1 and 2. Similar considerations apply to Table 19, which represents the instance of the nested relation `Priors`. In fact, each course is associated with all its preceding courses by means of the complex attribute `prerequisites`. □

4.1 Nested Relational Algebra

A meaningful amount of work on the extended relational model was devoted to the definition of query languages. Almost all paradigms of languages for the relational model have been extended to the nested case. The advantage of the extended relational model is here particularly evident: in principle it is not necessary to design new query languages, all is needed is the freedom to apply old constructs of relational query languages to more complex data structures. More formally, we need a language design that is fully *orthogonal*: since relations can now occur not only at the outermost level as external operands but even as complex attributes, the same operators should be applicable at the attribute level as well.

Due to the increased complexity of the data model, the algebraic approach has become more popular for the nested model than it was for the classical flat one, imposing itself as the predominant stream of research on languages for nested relations. An operational approach (as opposed to the declarative approach of the calculus-based and rule-based languages) is more appropriate, for

Table 19. Instance of the nested relation `Priors`

`course`	`prerequisites()` `prerequisite`
Compilers	Software engineering
Computers	Calculus
Geometry	Algebra
Operating systems	Programming languages
Programming languages	Computers
Robotics	Geometry Programming languages
Software engineering	Programming languages Databases

example, for specifying the schema-restructuring operations that are particularly relevant in the extended relational model.

This is accomplished through the introduction of two new algebraic operators, namely *nest* and *unnest*, defined later in this section, that allow creation and deletion of complex attributes starting from atomic domains.

Actually, the first proposals inherent to nested algebra involved these operators only, and did not discuss the extension of algebraic operators to the attribute level [22, 30, 42]. The idea was that, whenever relation-valued attributes are to be manipulated, one could first unnest, apply the standard relational operators and finally re-nest to obtain the desired result. However it was soon noticed that this cannot work in general, since unnest may not be reversible by nesting operations; moreover, this is neither an efficient nor a natural way of computing the results.

Subsequent research on algebraic languages for nested relations was then focused on investigating the expressive power of the algebra extended with the nest/unnest operators and on designing languages that were suited for efficient implementation and that allow for the manipulation of complex attributes without unnesting them first.

Algebras for manipulating complex attributes have been proposed by many authors [2, 17, 21, 29, 39, 40, 41]. In these algebras, ordinary algebraic operators are extended to accept nested relations as their operands. Certain algebras define set operators (union, difference, etc.) that apply recursively to all complex attributes of their operands.

The approach we follow in this section is to proceed from the more immediate and straightforward extensions to the more complex and specific ones. Specifically, we introduce the following class of extensions to relational algebra:

1. Set-theoretic operations and product extended to nested operands;
2. Nest and unnest;
3. Operators involving a predicate;
4. Extended projection;
5. Extended selection;
6. Expressions involving nested applications of operators.

Extension to set-theoretic operations and product. The basic operators on sets, namely union, difference, and intersection, are defined exactly as for flat relations: the only difference is that domains of attributes may now be either atomic or set-valued. The set operations are always performed at the most external level, that is, on tuples of external names, in other words, we cannot perform set operations on tuples of nested attributes.

Set-theoretic operations usually require operands to have the same domain. However, we can relax this requirement on relation schemas: the two schemas may also be equivalent (that is, isomorphic up to a renaming), if we give some rule to determine the names for the attributes of the result. To this end, we can introduce the following rule: when two schemas are equivalent but do not have the same attribute names, the result inherits the names of the first schema. Alternatively, an explicit renaming operator can be introduced as in the case of relational algebra.

Also the extension of the product is straightforward, since it only involves the extension of the schema at the external level.

Nest and unnest. These two operators produce a result obtained as a modification of both schema and instance of the operand. Informally, *nest*, denoted by ν, builds a higher-order attribute within an external relation starting from one or more atomic attributes, thus creating a further level of nesting. On the other hand, *unnest*, denoted by μ, deletes a higher-order attribute. When nesting, a set is created containing all the tuples of nested attributes having the identical values on the non-nested attributes. For unnesting, each tuple of the unnested attribute is concatenated with the external tuple containing the unnested attribute, thus resulting in a sort of tuple-oriented product. As remarked in [30], nest and unnest are actually the inverse of each other. However, while an unnest can always restore the situation previous to a nesting, the inverse is not in general true.

Unnest. The definition of μ can be formalized as follows. Given a database schema S, let r be a relation with schema R in S. Assume B is some higher-order name in E_{R} with an associated rule $B = (B_1, \ldots, B_m)$. Let $\{C_1, \ldots, C_k\} = E_{\mathrm{R}} - B$. Then

$$\mu_B(r) \tag{72}$$

is a relation r' with schema R' where:

Table 20. Result of the unnesting in Expression (73)

department	teachers()			
	name	age	course	year
Computer science	Angelique	42	Operating systems	4
	Angelique	42	Software engineering	4
	Carol	38	Databases	3
	Jerri	54	Compilers	5
	Jerri	54	Programming languages	3
Electronics	Patricia	36	Computers	2
	Richard	45	Robotics	5
Mathematics	Gregory	63	Calculus	1
	Martin	42	Geometry	2
	Sheila	40	Algebra	1

1. $R' = (C_1, \ldots, C_k, B_1, \ldots, B_m)$ and the rule $B = (B_1, \ldots, B_m)$ is removed from the set of rules in S if it does not appear in any other relation schema;
2. $r' = \{t \mid u \in r, t_{(C_1,\ldots,C_k)} = u_{(C_1,\ldots,C_k)}, t_{(B_1,\ldots,B_m)} \in u_{(B)}\}$.

Example 23. With reference to the nested relation `Departments` displayed in Table 18, the unnesting of collection `courses`:

$$\mu_{\texttt{courses}}(\texttt{Departments}) \tag{73}$$

is expected to yield the nested relation whose schema and instance are shown in Table 20. Owing to the unnest of `courses`, the result incorporates one tuple of `teachers` for each department. □

Nest. The definition of ν can be formalized as follows. Given a database schema S, let r be a relation with schema R in S. Let $\{B_1, \ldots, B_m\} \subset E_R$ and $\{C_1, \ldots, C_k\} = E_R - \{B_1, \ldots, B_m\}$. Assume that B does not occur in the left-hand side of any rule in S. Then

$$\nu_{B=(B_1,\ldots,B_m)}(r) \tag{74}$$

is a relation r' with schema R' where:

1. $R' = (C_1, \ldots, C_k, B)$, where rule $B = (B_1, \ldots, B_m)$ is appended to the set of rules in S;
2. $r' = \{t \mid u \in r, t_{(C_1,\ldots,C_k)} = u_{(C_1,\ldots,C_k)}, t_{(B)} = \{v_{(B_1,\ldots,B_m)} \mid v \in r, v_{(C_1,\ldots,C_k)} = t_{(C_1,\ldots,C_k)}\}\}$.

Example 24. To show the use of the nest operator, we show how we can generate the nested relation `Priors` displayed in Table 19 starting from the relation `Priorities` outlined in Table 3 as follows:

$$\texttt{Priors} \leftarrow \nu_{\texttt{prerequisites=(prerequisite)}}\texttt{Priorities} \ . \tag{75}$$

In the result, prerequisite courses relevant to the same course are grouped into the complex attribute `prerequisites` specified in the ν operator. □

Operators requiring a predicate. This class of operators includes selection and join. Predicates are more difficult to define in the extended relational model, due to the possibility of different nesting depths for the attributes. The problem can be illustrated as follows. Assume we have the following database schema: $\texttt{R} = (\texttt{A}, \texttt{M})$, $\texttt{M} = (\texttt{B}, \texttt{C})$, $\texttt{V} = (\texttt{D}, \texttt{N})$, $\texttt{N} = (\texttt{E}, \texttt{F})$, and consider the following operation:

$$\texttt{R} \bowtie_{\texttt{C=E}} \texttt{V} \ . \tag{76}$$

It is not clear at which level the tuples of the product of `R` and `V` should be selected. Specifically, should all the combinations of tuples in the inner collections be verified? Or the selection is taking place at the outermost level, selecting those tuples whose inner collections agree on the value of attributes `C` and `E` for all their tuples?

This semantical ambiguity (arising only in specific comparisons among attributes) depends on the (so called) *quantification level* of the attributes involved in the predicate. The general solution to this problem will be presented later. At the moment, we confine ourselves to extending the form of predicates in order to account for set-valued nested attributes. Thus, in this section, we consider selections and joins whose predicate contains attributes belonging only to the outermost level, that is, to the schema rule of the external name involved in the operation.

Extended propositional formula. Let R be a relation schema. A *propositional formula* $\wp$ over E_{R} is defined recursively as follows. *Atoms* over E_{R} have the form $A_1 \theta A_2$ or $A_1 \theta a$, where both A_1 and A_2 are in E_{R}, a is a constant, which can be set-valued, and θ is a comparison operator, that is, $\theta \in \{=, <, >, \neq, \geq, \leq, \supset, \supseteq, \subset, \subseteq, \in\}$. Every atom over E_{R} is a propositional formula over E_{R}; if $\wp_1$, $\wp_2$ are propositional formulas over E_{R}, then $\neg(\wp_1)$, $\wp_1 \wedge \wp_2$, and $\wp_1 \vee \wp_2$ are formulas over E_{R}. Parentheses can be used as usual. Nothing else is a formula. A propositional formula associates a Boolean value with each tuple in the instance r of R.

Selection. Given a relation r over the schema R, the *selection* of r with respect to $\wp$, denoted by $\sigma_{\wp}(r)$, is a relation over the same schema R, containing the tuples of r that make $\wp$ true:

$$\sigma_{\wp}(r) \stackrel{\mathrm{def}}{=} \{t \in r \mid \wp(t)\} \ . \tag{77}$$

The only changes introduced with respect to relational algebra are set comparison.

Example 25. With reference to the nested relation `Priors` displayed in Table 19, to find out the courses which require `Programming languages` as a prerequisite we write:

$$\sigma_{\text{`Programming languages'} \in \texttt{prerequisites}}(\texttt{Priors}) \ . \tag{78}$$

where '`Programming languages`' is a constant. This query will select the fourth, sixth , and seventh tuple of `Priors`. □

Join. The theta-join between nested relations can be defined based on Equivalence (10), that is, as a selection of a product:

$$r_1 \bowtie_{\wp} r_2 \ \equiv \ \sigma_{\wp}(r_1 \times r_2) \ . \tag{79}$$

where r_1 and r_2 are nested relations and $\wp$ an extended propositional formula, as defined above. The extension of the natural join to nested relations comes without any surprise, once considered that the implicit predicate of equalities between homonymous attributes may in general involve complex attributes.

Extended projection. The operators introduced so far, with the exception of nest and unnest, are a direct extension of relational algebra, but their expressive power is inadequate, since they lack the capability of accessing inner collections. In order to perform even simple queries, a lot of nesting and unnesting is therefore needed.

Example 26. Considering `Departments` in Table 18, to select the departments which refer to (at least) a course of the last two years, we cannot access directly attribute `year` in the selection predicate since it belongs to the inner relation `courses`. Thus we need to unnest in cascade `courses` and `teachers` so as to apply the selection on the resulting flat relation:

$$\pi_{\texttt{department}}(\sigma_{\texttt{year} \geq 4}(\mu_{\texttt{teachers}}(\mu_{\texttt{courses}}(\texttt{Departments})))) \tag{80}$$

□

The solution to this problem is to further extend relational operators by allowing the manipulation of inner collections. We first introduce projection so as to make it possible to project also on inner collections. Then we extend set operations so that they can perform union, difference, and intersection of inner attribute

Given an external name R, consider the set S' of rules corresponding to its relation schema. The projection $\pi_{A_1,\ldots,A_n}(R)$ defines the projection list $A_1, \ldots, A_n$: each name A_i must occur in the right-hand side of just one rule in S'. This is called the uniqueness constraint. The result of the projection has a relation schema S'' obtained by replacing the rules of S' with their projection on the projection list as follows:

1. Include in S'' the rules of S' that contain in their right-hand side one or several names of the projection list, limiting the right-hand side to names of the projection list;

2. Include in S'' the rules of S' that have on the left-hand side a name appearing in the rules of S'' that does not already occur on the left-hand side of a rule in S'';
3. Include in S'' the rules of S' that contain in their right-hand side a name appearing in S'', limiting the right-hand side to names appearing in S'';
4. Apply Steps 2 and 3 until no more rules are added to S''.

The instance r' of the result is obtained from the instance r of R by projecting each tuple on the new relation schema S''

We have a problem with the formalism: the 'projected' rules are valid for the result of the projection, but they are also used in the operand schema, so we should keep also the original, non-projected version. The simplest solution is to assume that, as in the flat relational model, each algebraic operation builds a temporary (nested) relation, that can be used as the operand of another operation or stored as a persistent relation. In both cases, the schema of the relation is defined by new rules, that are added and not substituted to the original version. Names should be changed accordingly, so as to respect the uniqueness constraint.

Example 27. With reference to `Departments` outlined in Table 18, the following projection:

$$\pi_{\texttt{department,name,courses}}(\texttt{Departments}) \tag{81}$$

will result in the nested relation shown in Table 21. □

Extended selection. In our previous version of the selection operator of nested relational algebra, selection was only possible at the outermost level, the level of tuples of the external relation. Informally, in order to be able to use inner attributes in the predicate, we need a way to decide:

1. On which collection the selection is performed;
2. Which values should be compared in a predicate atom.

We first formalize the concept of a *quantification level*, informally introduced above: constants and external names have quantification level 0 (that is, there is only one instance for them in the database); the quantification level of each name occurring in the right-hand side of a rule, for which the left-hand side name has quantification level i, is $i + 1$.

We now extend the notion of an atom in a selection predicate. Let R be a relation schema; atoms over E_{R} have the form $A_1 \theta A_2$ or $A_1 \theta a$, where a is a constant, θ is a comparison operator, $\theta \in \{=, <, >, \neq, \geq, \leq, \supset, \supseteq, \subset, \subseteq, \in\}$, and A_1, A_2 are names occurring in the relation schema such that:

(α) The quantification level of $A_1 \theta a$ is that of the parent of A_1 and the atom refers to its tuples;
(β) If A_1 and A_2 are siblings (they have a common parent) in G_{R} (the schema tree corresponding to R), then the quantification level of $A_1 \theta A_2$ is that of the common parent, and the atom refers to its tuples;

Table 21. Result of the extended projection (81)

department	teachers()	
	name	courses()
		course year
Computer science	Angelique	Operating systems 4 Software engineering 4
	Carol	Databases 3
	Jerri	Compilers 5 Programming languages 3
Electronics	Patricia	Computers 2
	Richard	Robotics 5
Mathematics	Gregory	Calculus 1
	Martin	Geometry 2
	Sheila	Algebra 1

(γ) If a sibling of A_1 is an ancestor of A_2 (or vice versa) in G_{R}, then the quantification level of $A_1 \theta A_2$ is that of the parent of A_2 (respectively A_1) and the atom refers to its tuples.

According to the above axioms, external constants and external names are considered as siblings of any external name (the quantification level is 0).

Propositional formulas of predicates are built by means of atoms. For each atom involved in a propositional formula, we consider the tuples to which the atom refers. Operands of binary logical connectives must be atoms or other formulas whose corresponding tuples still obey to the above axioms. The logical connective will refer to tuples of the lower quantification level, unless they belong to the same parent collection.

Therefore, we have given axioms to recursively determine the quantification levels and tuples to which the predicate refers; the selection is performed only on these tuples.

Example 28. Based on the above axioms, the following extended selection:

$$\sigma_{\texttt{department}=\text{`Computer science'} \wedge \texttt{age}>40}(\texttt{Departments}) \tag{82}$$

will (somewhat surprisingly) result in the nested relation displayed in Table 22. With reference to Axioms (α), (β), and (γ), the quantification level of the selection predicate $\wp$ within Expression (82) is computed as described in Table 23.

Table 22. Result of the extended selection (82)

department	teachers()			
	name	age	courses()	
			course	year
Computer science	Angelique	42	Operating systems	4
			Software engineering	4
	Jerri	54	Compilers	5
			Programming languages	3
Electronics				
Mathematics				

Table 23. Steps to yield the quantification level of the selection predicate $\wp$ in (82)

Predicate	Description	Axiom	Quantification level
$\wp_1$	department = 'Computer science'	(α)	Departments
$\wp_2$	age > 40	(α)	teachers
$\wp$	$\wp_1 \wedge \wp_2$	(γ)	teachers

Accordingly, tuples of teachers are selected based on the value of predicate $\wp$. Within Table 22, the nested attribute teachers is empty for both departments Electronics and Mathematics, because every tuple of them does not satisfy predicate $\wp_1$ and, consequently, $\wp$. Besides, removing a tuple from teachers causes the removal of all its attributes, specifically, courses. □

Note that, generally speaking, when the quantification level of a selection is internal, that is, the selection is applied to an internal relation R (in Example 28, R = teachers), duplicates may be generated among the tuples R belongs to (in Example 28, the tuples of Departments). In fact, in this case, the selection operates as a modification of attribute values, which is bound to generate duplicates.

Expressions involving nested applications of operators. Since operands are nested relations, it is worth nesting the operators. For example, the selection predicate could consist of a comparison between the results of the selections on two inner attributes. This way, relational operators can be applied to inner attributes as well, thereby extending the expressiveness of the algebra.

Nested expressions are used also in languages for the flat relational model: for example, SQL allows nested queries, that are generally used as an alternative to joins between relations. When the structure of relations is more complicated, the need for nested expressions becomes a natural requirement.

Nested expressions in the context of extended algebras were first studied in [29] (where they are called *recursive* expressions) and [41].

Essentially, nested algebraic operations can be defined on the basis of the following principles:

1. A relational expression can occur wherever a relation name is needed (this was implicitly assumed in some of our previous examples).
2. A relational expression can occur wherever an attribute name is expected.

Example 29. To find out the departments that include (at least) a teacher younger than 40 we may write:

$$\pi_{\texttt{department}}\left(\sigma_{(\sigma_{\texttt{age}<40}(\texttt{teachers}))\neq\emptyset}(\texttt{Departments})\right) \tag{83}$$

In Expression (83), a selection is applied to `Departments`, whose predicate involves a complex comparison between a nested selection on `teachers` and the empty set. Intuitively, each tuple of `Departments` is selected if and only if the selection of the corresponding attribute `teachers` based on the simple comparison $\texttt{age} < 40$ yields at least a tuple, in other words, when the department includes a teacher younger than 40. This holds for `Computer science` and `Electronics` only, which are in fact the departments displayed after the final projection. □

Example 30. To find out the departments that include (at least) a teacher older than 50 and teaching a course of the last two years we may write:

$$\pi_{\texttt{department}}\left(\sigma_{\left(\sigma_{\texttt{age}>50\wedge(\sigma_{\texttt{year}\geq 4}(\texttt{courses})\neq\emptyset)}(\texttt{teachers})\right)\neq\emptyset}(\texttt{Departments})\right) \tag{84}$$

In Expression (84), a selection is applied to `Departments`, whose predicate involves a complex comparison between a nested selection on `teachers` and the empty set. Besides, the nested selection on `teachers` involves a complex comparison between a nested selection on `courses` and the empty set. The latter selection is based on the simple comparison $\texttt{year} \geq 4$. This way, the selection operation is propagated through the nested schema of the operand `Departments`. The final result is expected to include the singleton $\{(\texttt{Computer science})\}$. □

5 Multisets in Nested Relational Database Systems

In Sect. 3, we showed how the relational model may be extended to deal with multisets. In particular, in Sect. 3.1, the operators of relational algebra have been extended to manipulate multisets. Then, in Sect. 4 we showed how the relational model can be extended to incorporate nested relations and how these nested relations can be manipulated by means of a nested relational algebra.

It is natural at this point to combine the two extensions, namely multisets and nested relations, into a new data model in which multisets can be nested

within *complex relations*. Intuitively, a complex relation is like a nested relation incorporating both sets and multisets[4].

We may easily extend the formal definition of a nested relational database schema given in Sect. 4 to a complex relational database schema by making rules polymorphic, that is, a rule R may be either

$$R = (R_1, R_2, \ldots, R_n) \tag{85}$$

or

$$R = [R_1, R_2, \ldots, R_n] \tag{86}$$

where, intuitively, Rule (85) defines a set, while Rule (86) a multiset.

Example 31. Shown in Table 24 is a complex relation called `Products`, whose schema is defined by the following rules:

$$\begin{array}{lcl} \texttt{Products} & = & (\,\texttt{product}, \texttt{components}, \texttt{time}\,) \\ \texttt{components} & = & [\,\texttt{component}\,] \end{array} \tag{87}$$

Each `product` is associated with a multiset `components`[5] and a `time` needed to assemble the product by means of the collection of components. Thus, according to the simplified information stored in `Products`, a `car` is composed of a `body`, an `engine`, four occurrences of `wheel` , two occurrences of `light`, and a `tank`. Then, each component is itself a product, possibly composed of other components, and so on, until reaching atomic components, such as `light` or `tank`. □

5.1 Nested Relational Algebra for Complex Relations

Intuitively, we may define a nested relational algebra for complex relations by combining the concepts inherent to the relational algebra for multisets introduced in Sect. 3.1, and the relational algebra for nested relations presented in Sect. 4.1. In fact, there is almost nothing surprising in integrating the two approaches into a uniform operational framework. Among other papers, the book [10] presents an extensive description of a nested relational algebra for complex relations. However, in this section, we only focus upon a limited variety of computational aspects relevant to the manipulation of complex relations.

Type transformation. Generally speaking, a complex relation is polymorphic in nature, as it may be either a set or a multiset and include both set and multiset attributes. For example, the complex relation `Products` shown in Table 24 is a

[4] Complex relations may include other kinds of collections, like *lists* and *arrays*. However, according to the scope of this paper, we confine our discussion to multisets only.

[5] Within the schema, the name of a multiset, such as `components`, is followed by square brackets to distinguish it from a set.

Table 24. Instance of the complex relation `Products`

`product`	`components[]` `component`	`time`
car	body engine wheel wheel wheel wheel light light tank	32
body		0
engine	block block carburetor	18
block	cylinder piston valve	26
wheel	tyre tube	4
light		0
tank		0
carburettor		0
cylinder		0
piston		0
valve		0
tyre		0
tube		0

set, but the internal collections relevant to attribute `components` are multisets. That is, the schema of `Products` involves both set and multiset constructors. Since these two collection types may coexist within the same complex relation, it makes sense to define an operator which allows us to change the set into a multiset and vice versa[6].

[6] Such a change, however, must always be interpreted within a functional framework, where operands never change their state but, rather, a new complex relation is generated with the changed schema.

Let r be a complex relation, and $\ell = A_1, \ldots, A_n$ a (possibly empty) set of attribute names of r. Then, the following expression:

$$\tau_\ell(r) \tag{88}$$

denotes the type transformation of r, which results in a complex relation r' such that:

1. If $\ell = \emptyset$, then the type of r' is changed with respect to the type of r;
2. If $\ell \neq \emptyset$, then the types of attributes of r' in ℓ are changed with respect to the corresponding types in r;
3. The instance of r' is obtained from the instance of r in accordance with the type transformations performed in Points 1 and 2.

Example 32. With reference to Table 24, to generate a new complex relation obtained by transforming `Products` into a multiset we may write:

$$\tau(\texttt{Products}) \ . \tag{89}$$

Instead, to change the nested multiset `components` into a set we write:

$$\tau_{\texttt{components}}(\texttt{Products}) \ . \tag{90}$$

Consequently, the duplicates originally contained in multiset `components` (see Table 24) will be removed from the result. □

Type transformations are not only useful but even necessary to specify queries appropriately. In set-theoretic operations, since we require the equality of the operand schemas, a renaming might be in general not sufficient if one operand is a set and the other a multiset. Moreover, we require as well that product and join operate on complex relations of the same type. Thus, type transformation might be needed in this case too[7].

Nest and unnest. Nest and unnest operations were introduced in Sect. 4.1 for nested relations. Instead of extending such operations formally, we prefer providing some intuitive guidelines.

Unnest. The unnest operation creates a complex relation by reducing by one level the depth of the operand schema tree. Given an instance r of a complex relation schema R defined as follows:

$$R\langle A_1, \ldots, A_m, \beta\langle B_1, \ldots, B_n\rangle\rangle \tag{91}$$

[7] Another viable approach might be to have implicit casting rules (as in general-purpose programming languages in which, for example, an integer is transformed into a real), where the type of a complex relation is automatically transformed into another type based on the specific computational context.

Table 25. Result of Expression (94)

product	component	time
car	body	32
car	engine	32
car	wheel	32
car	light	32
car	tank	32
engine	block	18
engine	carburetor	18
block	cylinder	26
block	piston	26
block	valve	26
wheel	tyre	4
wheel	tube	4

where angles denote either set or multiset collections, the following operation:

$$\mu_{\beta}(r) \tag{92}$$

generates a complex relation r' with the following schema R':

$$R'\langle A_1, \ldots, A_m, B_1, \ldots, B_n \rangle \tag{93}$$

Since more than one tuple on $B_1, \ldots, B_n$ exists in R for each tuple $A_1, \ldots, A_m$, r' will be built joining the tuples together. The type of R' is determined from the type of R and not from the type of β. Accordingly, the tuples resulting from the concatenation of the tuples of R and β will be considered as belonging to a complex relation of the type of R. Two cases are possible:

1. R is a set: the multiple occurrences of the tuples on β will be reduced to one;
2. R is a multiset: the multiple occurrences of the tuples of β will generate multiple occurrences in r' too.

Example 33. With reference to the complex relation `Products` shown in Table 24, the following operation:

$$\mu_{\texttt{components}}(\texttt{Products}) \tag{94}$$

will result in the (flat) set displayed in Table 25. Note that the unnesting of empty collections give rise to the removal of the corresponding tuple. □

Nest. Given an instance r of a complex relation schema R defined as follows:

$$R\langle A_1, \ldots, A_m, B_1, \ldots, B_n \rangle \tag{95}$$

where each A_i and B_i can be a complex attribute, the following operation:

$$\nu_{\beta=\langle B_1,\ldots,B_n\rangle}(r) \tag{96}$$

generates a complex relation r' with the following schema:

$$R'\langle A_1, \ldots, A_m, \beta\langle B_1, \ldots, B_n\rangle\rangle \ . \tag{97}$$

The instance of r' is built as follows: tuples having the same projection on attributes $A_1, \ldots, A_m$ are grouped together and for each group we consider the corresponding parts B_i which form a set or a multiset, according to the type of β. Two cases are possible:

1. R is a set: only one tuple is maintained in r', which is obtained through the concatenation of the A_i, $i \in [1\,..\,m]$, and β;
2. R is a multiset: a tuple of r' is created for each tuple of r; within the tuples of the same group the part β is repeated.

Example 34. Shown in Table 26 are a multiset `Flat` (left) and the result of the following nest operation (right):

$$\nu_{\texttt{components=(component)}}(\texttt{Flat}) \ . \tag{98}$$

Expression (98) generates a complex relation of type multiset with a nested set called `components`. Therefore, according to the above rules, for each tuple in `Flat`, a tuple is created in the result, where, within the tuples of the same group, the instance of the newly created attribute is repeated. Note, however, that, since `components` is defined as a set, duplicates are removed from the nested collections. □

Table 26. Multiset `Flat` (left) and result of Expression (98) (right)

product	component
car	wheel
car	wheel
car	engine
engine	block
engine	block
engine	carburetor

product	components() component
car	wheel engine
car	wheel engine
car	wheel engine
engine	block carburetor
engine	block carburetor
engine	block carburetor

Table 27. Result of Expression (101)

component
body
engine
wheel
light
tank
block
carburetor
cylinder
piston
valve
tyre
tube

Complex aggregate operations. We have already introduced the notion of an aggregate operation in Sect. 3.4 within the context of SQL. Considering complex relations, it is possible to introduce new kinds of aggregate operations that are applied to collections of complex (rather than simple) values. Specifically, these are *complex union* and *complex intersection*, denoted by $\cup^{\mathrm{c}}$ and $\cap^{\mathrm{c}}$, respectively.

Let r be a complex relation involving an attribute A, either of type set or multiset[8]. The complex union of A within r is defined as follows:

$$\cup^{\mathrm{c}}_{A}(r) \stackrel{\mathrm{def}}{=} \cup_{t \in r}(t_{(A)}) \ . \tag{99}$$

Likewise, the complex intersection of A within r is so defined:

$$\cap^{\mathrm{c}}_{A}(r) \stackrel{\mathrm{def}}{=} \cap_{t \in r}(t_{(A)}) \ . \tag{100}$$

Note that Definitions (99) and (100) can be generalized to deal with an attribute A which is at any level within the schema of r, not necessarily at the level of the tuples of r.

Example 35. With reference to the complex relation `Products` displayed in Table 24, we might ask for the set of products that are components of some other product as follows:

$$\tau(\cup^{\mathrm{c}}_{\mathtt{components}}(\mathtt{Products})) \tag{101}$$

which gives rise to the set displayed in Table 27. Since both union and intersection of multisets result in a multiset, the final type transformation is needed in order to remove duplicated components.

Note that, if we replaced in (101) $\cup^{\mathrm{c}}$ with $\cap^{\mathrm{c}}$, we would obtain the empty set. □

[8] Attribute A can be itself complex, that is, it may be defined in terms of other complex attributes.

```
function FixedPoint(r): relation resulting from φ[Δ ← F_t(r); F_p(Δ, r); F_c(Δ, r)](r)
  input
    r: the operand relation;
  begin
    r' := r;
    stop := false;
    repeat
      Δ := F_t(r');
      if F_p(Δ, r') then
        stop := true
      else
        r' := F_c(Δ, r')
    until stop;
    return r'
  end.
```

Fig. 3. Semantics of the fixedpoint operation defined in Formula (102)

Fixedpoint operation. An interesting extension to the algebra for complex relations is the *fixedpoint* operator, which allows recursive queries to be expressed in an algebraic style without the use of control structures or recursion. Such an operation can be seen as a generalization of the *closure* operation introduced in various advanced query languages.

The fixedpoint operation applies repeatedly a function $\mathcal{F}_\mathrm{t}$, called the *transformer*, to a complex relation r, combining the result with r using another function $\mathcal{F}_\mathrm{c}$, called the *combiner*, which generates the new relation to be used in place of r in the next iteration. The loop continues until the *predicate* $\mathcal{F}_\mathrm{p}$ becomes true.

From the algebraic point of view, the fixedpoint operator, denoted by φ, is a unary operator computing a result which has the same schema of the operand (but, in general, different instance). The generic form of the fixedpoint operation applied to a complex relation r is the following:

$$\varphi\,[\,\Delta \leftarrow \mathcal{F}_\mathrm{t}(r);\ \mathcal{F}_\mathrm{p}(\Delta, r);\ \mathcal{F}_\mathrm{c}(\Delta, r)\,]\,(r) \tag{102}$$

where the involved functions $\mathcal{F}_\mathrm{t}$, $\mathcal{F}_\mathrm{c}$, and $\mathcal{F}_\mathrm{p}$ are specified within square brackets, between the operator symbol and the operand. The semantics of the fixedpoint operator can be specified by the pseudo-code of a *FixedPoint* function, which takes as input the operand r and computes the result r', as illustrated in Fig. 3. The meaning is the following: r is the name of the φ operand and r' is a copy of r. First, r' is taken as the argument of the transformation function $\mathcal{F}_\mathrm{t}$ (possibly with other arguments) that calculates the partial value Δ. Before going to the combination function, the predicate $\mathcal{F}_\mathrm{p}(\Delta, r')$ is calculated. This predicate may possibly contain further arguments beside Δ and r'. If the predicate evaluates to true, the loop is broken and the current instance of r' is the result.

The quantification level of the predicate $\mathcal{F}_\mathrm{p}$ must be related with the quantification level of the operand according to the axioms given in Sect. 4.1. If the predicate $\mathcal{F}_\mathrm{p}(\Delta, r')$ evaluates to false, the result of the combination function

Table 28. Query result of Example 36

product
cylinder
cylinder
cylinder
cylinder
piston
piston
piston
piston
valve
valve
valve
valve
carburetor
carburetor
tyre
tube

$\mathcal{F}_{\mathrm{c}}(\Delta, r')$, possibly with further arguments, is assigned to r', and the body of the loop is repeated [9]. When the loop terminates, the instance of r' is the returned result.

Example 36. Consider the complex relation `Products` displayed in Table 24. We might be interested in finding out the multiset of atomic components needed to assemble a multiset of given products. For example, to assemble two engines we need four cylinders, four pistons, four valves, and two carburetors.

The multiset containing the products for which we ask atomic components is called `Input`, which is characterized by a single attribute called `product`. For example, `Input` might include two engines and one wheel. Here is our query:

$$\texttt{Input} \leftarrow [(\texttt{engine}), (\texttt{engine}), (\texttt{wheel})]$$

$$\begin{aligned}\varphi[\ & \texttt{Comps} \leftarrow \rho_{\texttt{product}}(\cup^{\mathrm{c}}_{\texttt{components}}(\texttt{Input} \bowtie (\tau(\texttt{Products})))); \\ & \texttt{Comps} = \emptyset; \\ & \texttt{Comps} \cup (\pi_{\texttt{product}}(\sigma_{\texttt{components}=\emptyset}(\texttt{Input} \bowtie (\tau(\texttt{Products}))))) \\ ]&(\texttt{Input})\end{aligned}$$

The first statement simply instantiates the multiset `Input`. The actual query is expressed by the next expression, where the fixedpoint operator is applied to `Input`. According to the terminology introduced in Formula (102), the transformer first makes a natural join between `Input` and the multiset mirror of `Products`, then makes the complex union of the relevant components, and finally performs a renaming of the result. This way, `Comps` is expected to contain

[9] Note that the function will not terminate if the condition $\mathcal{F}_{\mathrm{p}}(\Delta, r')$ never evaluates to true.

the multiset of components (at the first level) of the products in `Input`. The predicate tests for the emptiness of `Comps`. Then, the combiner makes the union of `Comps` and the atomic products contained in the current instance of `Input`. The result of φ is displayed in Table 28. □

6 Related Work

Various database systems and query languages support multisets in the relevant data model [15, 47, 28, 45, 36, 44, 23, 12, 10]. For instance, a query written in DAPLEX [45] supports a flexible control over creation and elimination of duplicates in the query result. Therefore, starting from the 1980s, multisets were proposed as an additional type constructor in order to formally deal with duplicates in databases.

Some research has been devoted to extending the set-theoretic operations to multisets, as well as to developing techniques for algebraic query optimization for data models supporting multisets. The work presented in [20] extends the relational model to include *multiset relations*, i.e. relations with duplicated tuples. A framework for query optimization is also defined. Both the operands and the results of queries are in general multiset relations.

A different approach to formalize multisets in relational databases can be found in [31], wherein, at the conceptual level, a relation with duplicates, called *multirelation*, is conceived as a subset of the columns of a larger relation with no duplicates, by keeping hidden those columns that yield the uniqueness of all the tuples in the relation. Consequently, relations or views with duplicated tuples are not allowed at the conceptual level, and there is no need for the introduction of any explicit multiset semantics.

In [27] it is investigated how the use of multisets in the nested relational algebra extends its expressive power and increases its computational complexity.

In [49] a framework for algebraic query optimization is developed in the context of an object-oriented data model that supports multisets. In the quoted work a number of specific algebraic rules for query transformation are presented. The emphasis is posed on rules for manipulating the instances of types in a supertype/sub-type lattice, with complex type constructors. Union, intersection, and difference are defined for multisets, and some transformation rules for these operators are given.

The work described in [38] deals with recursion and aggregates according to a proposed declarative multiset semantics for deductive databases. Such databases have been enriched with aggregate operators and predicates that may return multisets of tuples. A formal basis for efficient evaluation of queries when multisets are generated as intermediate results in a query evaluation process is provided.

6.1 Multisets and Views

Given a relational database, views are data derived from it that can be materialized (that is, stored permanently in the database) and subsequently queried

against. The use of views in order to answer queries is common to different database areas and it is very relevant in data warehouse applications [32]. In fact, in such applications, queries against data typically involve the computation of aggregate operations, which, as known, have to handle duplicated data to be correctly evaluated.

In [46] the problem of transforming an SQL query that includes grouping and aggregation into an equivalent one, that can be answered more efficiently, is considered. Such a rewriting has to fulfill two requirements:

1. The rewritten query has to be posed against available materialized views (conditions under which this is possible are given);
2. The multiset semantics of queries has to be preserved, that is, the rewritten query computes the same multiset of tuples for any given database.

The problem of view maintenance occurs when some of the base relations change and, consequently, materialized views must be recomputed to ensure that answers to queries posed against the views be correct. The recomputation of a view, after a change has occured in base relations, is usually computationally expensive. An alternative approach to complete recomputation consists in determining the changes that must be applied to the view, given the changes to base relations and the expression that define the view. In [25] an algorithm for view maintenance that follows this alternative approach is proposed. Views are regarded as multisets of tuples defined by means of expressions of a multiset algebra. The algorithm for view maintenance gives solutions that satisfy some minimality requirements, and, moreover, that are proven to be correct with respect to the aggregate operations computed over the changed view.

6.2 Multisets in Web Information Discovery

Web data mining [35] is defined as the discovery and analysis of useful knowledge from Web data. An approach to Web data mining is proposed in [7], according to which queries are posed against the Web by specifying an entry point Web address and some conditions about paths and data attributes. Once retrieved, data matching the query are stored in a special relation. Each tuple of such a relation consists of interconnected directed graphs, which represent Web documents and hyperlinks. In order to isolate the columns of interest from a table, a *Web projection* operator can be used, which does not eliminate identical tuples automatically. The result of a Web projection is a *Web bag*, that is a relation that may contain multiple occurences of the same tuple. A few differences between Web bags and multisets in relational databases can be identified:

1. Different nature of data: a Web bag contains unstructured data derived from the Web, while each piece of data in a multiset within a relational database has a known domain;
2. Different purposes: Web bags are useful for discovering knowledge, while multisets in relational databases mainly focus on efficiency and on semantics of aggregate operators;

3. Different origin: Web bags can only be obtained as the result of a projection, while multisets in relational databases can be obtained also as the result of other operations.

The exploitation of Web bags for discovering useful knowledge from Web data is discussed in the same contribution.

6.3 Query Languages for Multisets

In this section we will describe in some detail two meaningful works in the area of database query languages for multisets. The first [5] is a conference paper that historically opened the way to a research stream. The second work [33, 26, 34], carried out over several years, moved many further steps in this stream. In both works the name *bag* is preferred rather than *multiset* and we will adhere to this preference. In each description we will retain the notation used by the authors of the considered works. Different authors adopt different notations, that is, they may use different symbols for denoting the same operator or, vice versa, they may use the same symbol for denoting distinct operators. Thus, when faced with algebraic properties, the reader has to recall the context-specific meaning of the involved operators. In order to help the reader, while providing any operator definition given in a work, we have explicitly stated the correspondence of the current operator with operators defined by other authors and/or introduced in previous sections.

Algebraic properties of bags. Albert [5] defines some bag operations and investigates to what extent the typical algebraic properties of set operations are obeyed when bags are assumed as operands. The proposed semantics of operators agree with [20]. Nested bags are not dealt with. The ultimate goal is to provide a formal basis for database query optimization.

Constants. The notation [*list of elements*] is used for representing a bag, where each element belongs to a countable set of primitive objects. The symbol $\emptyset$ denotes an empty bag.

Membership and multiplicity. The expression $x \in B$ denotes that bag B contains x, and $x \in\in B$ represents the number of copies of x in B. If $x \in B$, then $x \in\in B > 0$, and vice versa. For example, given bag $B = [a, a, b, b, b]$, both $a \in B$ and $b \in B$ hold; in particular, $a \in\in B$ equals 2, and $b \in\in B$ equals 3.

Containment. Given two bags A and B, by definition

1. $A \subseteq B$ means that $(\forall x \in A)\ ((x \in\in A) \leq (x \in\in B))$;
2. $A \subset B$ means that $A \subseteq B$ and $A \neq B$.

In either case, A is a *subbag* of B. A theorem is proven, according to which $\subset$, as defined above, is a partial order relation (i.e. it is an irreflexive transitive relation) on any set of bags.

Powerset. Given a bag A, by definition, the powerset of A is

$$\mathcal{P}(A) \stackrel{\text{def}}{=} \{B \mid B \subseteq A\} \ . \tag{103}$$

Union and intersection. Let A and B be bags. By definition

1. $A \cup B$ is the smallest bag C such that $A \subseteq C$ and $B \subseteq C$;
2. $A \cap B$ is the largest bag C such that $C \subseteq A$ and $C \subseteq B$.

The notion of *largest* and *smallest* used in the above definitions refer to the partial order relation $\subset$.

The definitions of $\cup$ and $\cap$ yield the usual set union and intersection when restricted to sets. Besides, these operations, as defined on bags, share the same algebraic properties as the standard set-theoretic ones. This is proven by means of a theorem which states that, given a bag U, ($\mathcal{P}(U)$,$\cup$, $\cap$) is a distributive lattice, called *subbag lattice*, induced by the partial order $\subset$. This means that, for the binary operations $\cup$ and $\cap$, as applied to any possible subbag of a given (universal) bag U, all the axioms of Boolean algebra wherein complement is not referenced hold, with U corresponding to one and $\emptyset$ to zero.

Note that the union operation, as defined above for bags, is neither that defined in Sect. 3.1 nor that adopted by SQL for multisets. As to the intersection operator, it corresponds to both the intersection as defined in Sect. 3.1 and the SQL operator `INTERSECT ALL` (see Sect. 3.4).

By means of a theorem, it is shown that the above operations are well-defined, that is, for any given pair of bags, there is a unique bag satisfying the definition of union/intersection of such bags. In particular, given three bags A, B, and C, for union it is proven that

1. $x \in\in A \cup B = \max(x \in\in A, x \in\in B)$;
2. $A \subseteq C$ and $B \subseteq C \Rightarrow A \cup B \subseteq C$.

Likewise, for intersection it is proven that

1. $x \in\in A \cap B = \min(x \in\in A, x \in\in B)$;
2. $C \subseteq A$ and $C \subseteq B \Rightarrow C \subseteq A \cap B$.

Concatenation. Given two bags A and B, and the countable set of primitive objects Obj that is the domain of any element of A and B, by definition the concatenation of A and B, $A \sqcup B$, is the bag C such that

$$x \in\in C = (x \in\in A) + (x \in\in B) \ . \tag{104}$$

This operator becomes the usual set union when bags are turned into their set counterparts. SQL use the keyword `UNION ALL` for this operator.

Difference. Let A and B be bags, and Obj the countable set of primitive objects that is the domain of any element of A and B. By definition, the difference $A \setminus B$ is the bag $C \subseteq A$ such that

$$(\forall x \in Obj)\ (x \in\in C = \max((x \in\in A) - (x \in\in B), 0)) \ . \tag{105}$$

This operator matches the usual notion of set difference when restricted to sets. It corresponds to the difference operator for multisets defined in Sect. 3.1 as well as to the `EXCEPT ALL` operator of SQL (see Sect. 3.4). However, taking any bag U that is not a set as a universe, this operator, if used as a unary operator, does not induce a complement operator relative to this universe. Operator $\setminus$ would be a unary complement operator if it would satisfy the following two axioms:

$$\begin{array}{l}(\forall A \in \mathcal{P}(U))\ (A \cup (\setminus A) = U) \\ (\forall A \in \mathcal{P}(U))\ (A \cap (\setminus A) = \emptyset)\ .\end{array} \tag{106}$$

However, these two axioms does not hold for operator $\setminus$ as defined above. Still more, Albert demonstrates a theorem according to which no such an operator exists and, therefore, looking for a different semantics for bag difference cannot solve the problem.

Given a complement operator defined on a universe and two operators that form a distributive lattice on the same universe, the three operators altogether are a Boolean algebra. Thus $(\mathcal{P}(U), \cup, \cap, \setminus)$ is not a Boolean algebra for bags, whereas it is for sets. Therefore, while all the properties of a Boolean algebra which involve the difference operator hold for sets, there are some of them which fail for bags, as illustrated in the following example, taken from [5].

Example 37. Let's consider the following algebraic transformations inherent to the set-theoretic operations of union, intersection, and difference:

$$A \setminus (B \cup C) = A \cap (\overline{B \cup C}) = A \cap (\bar{B} \cap \bar{C}) = (A \cap \bar{B}) \cap \bar{C} = (A \setminus B) \setminus C\ . \tag{107}$$

The equivalence of set difference and the intersection with the complement has been exploited in the first and last transformations, in the second transformation De Morgan's law has been applied, and, in the next one, the associativity of intersection.

The identity obtained between the initial and final expressions holds if A, B, and C are sets, however, in general, it does not hold for bags. In fact, let's suppose, for instance, that $A = [x, x, x, x]$, $B = [x, x]$, and $C = [x]$. Then $A \setminus (B \cup C) = [x, x]$, while $(A \setminus B) \setminus C = [x]$. □

Duplicate elimination. A bag B is transformed into a set by function δ. By definition, $\delta(B) = \{x \mid x \in B\}$. δ looks like the opposite function of the mirror function $\mathfrak{M}$ defined in Sect. 3.2.

(Boolean) selection. Let A be a bag, Obj the domain of its elements, and ψ a quantifier-free predicate, either atomic or built up from a set of atomic predicates, whose domain includes $\delta(A)$. The selection of A based on ψ selects all and only the elements of A which satisfy the predicate ψ, that is, by definition, $\sigma_\psi(A)$ is the bag C such that

$$(\forall x \in Obj)\ \left(x \in\in C = \begin{cases} x \in\in A & \text{if } \psi(x) \\ 0 & \text{otherwise} \end{cases} \right)\ . \tag{108}$$

This operator is the same as the selection operator for multisets defined in Sect. 3.1, which can be expressed in SQL by means of the select-from-where paradigm (see Sect. 3.4).

A theorem [5] states that, given any predicates ψ and φ whose domains include $\delta(A)$, the following equalities hold:

$$\begin{aligned} \sigma_{\psi\vee\varphi}(A) &= \sigma_{\psi}(A) \cup \sigma_{\varphi}(A) \\ \sigma_{\psi\wedge\varphi}(A) &= \sigma_{\psi}(A) \cap \sigma_{\varphi}(A) \\ \sigma_{\neg\varphi}(A) &= A \setminus \sigma_{\varphi}(A) \\ \delta(\sigma_{\varphi}(A)) &= \sigma_{\varphi}(\delta(A)) \\ x \in\in \sigma_{\varphi}(A) &= (x \in\in A) \cdot (x \in\in \sigma_{\varphi}(\delta(A))) \ . \end{aligned} \tag{109}$$

The first three properties above imply that the semantics of $\vee$, $\wedge$, and $\neg$ with respect to Boolean selection correspond to $\cup$, $\cap$, and $\setminus$, respectively.

The next two properties formally state the *all-or-nothing* semantics of the selection operator, that is, either all copies of some $x \in A$ are selected, or none. The fourth property exhibits a slight abuse of notation on the right-hand side since the Boolean selection operator defined over bags is applied to a set. Indeed, Albert makes no clear distinction between a set and a bag containing just an occurrence of everyone of its element.

Albert proves a theorem stating that, with respect to selection operations, bags behave like sets. Formally, let E_1 and E_2 be two compositions of selection operators, that is, $E_1 = \sigma_{\psi_1} \circ \sigma_{\psi_2} \cdots \circ \sigma_{\psi_n}$ and $E_2 = \sigma_{\varphi_1} \circ \sigma_{\varphi_2} \cdots \circ \sigma_{\varphi_k}$. If, for every set A, $E_1(A) = E_2(A)$, then also $E_1(B) = E_2(B)$ for every bag B.

Complement. Let U be a bag and $Pred$ the smallest set of predicates that includes a given set of atomic predicates, and is closed under the propositional connectives. Let $S(U)$ be the set of subbags of U obtained by applying any selection operator to U, that is,

$$S(U) = \{A \subseteq U : (\exists\varphi \in Pred)(A = \sigma_{\varphi}(U))\} \ . \tag{110}$$

The unary complement operator $-$ is defined on $S(U)$. By definition, for any $A \in S(U)$, $-(A) = U \setminus A$. Albert proves that $(S(U), \cup, \cap, -)$ is a Boolean algebra.

Operators and query languages. By means of the following properties, Albert proves that $\sqcup$ and $\setminus$ are sufficient to obtain $\cup$ and $\cap$.

$$\begin{aligned} A \cup B &= (A \setminus B) \sqcup B \\ A \cup B &= (A \sqcup B) \setminus (A \cap B) \\ A \cap B &= A \setminus (A \setminus B) \\ A \cap B &= (A \sqcup B) \setminus (A \cup B) \ . \end{aligned} \tag{111}$$

The converse, instead, does not hold, that is, neither $\sqcup$ nor $\setminus$ can be constructed out of the three remaining operations, as stated by means of two theorems.

The implication of these results is that $\sqcup$ and $\setminus$ are strictly more general operations than $\cup$ and $\cap$. This justifies the presence in SQL of `UNION ALL` and

`EXCEPT ALL`, corresponding respectively to operators $\sqcup$ and $\setminus$, along with the absence of any operator corresponding to bag union $\cup$ and intersection $\cap$ (as defined by Albert).

Since a Boolean algebra $(S(U), \cup, \cap, -)$ has been defined on the set of the results of any Boolean selection as applied to some universal bag, the usual algebraic properties for sets hold within this domain. In particular, the algebraic transformations that are applied to sets for query optimization keep being valid when the operands are bags which are formed by applying a Boolean selection to some universal bag.

Properties of query languages for bags. The research described in [33, 26, 34] focuses on how database query languages are affected by the use of duplicates.

Many query languages have traditionally been developed based on relational algebra and relational calculus. Relational algebra is the standard language for sets, and (in its flat form) an algebraization of first-order logic. However, structures such as bags, along with nested structures in general, are not naturally supported by first-order logic-based languages. Thus, in the work at hand an approach to language design was exploited [11] that, instead of using first-order logic as a universal platform, suggests to consider the main type constructors (such as sets, bags, and records) independently and to turn the universal properties of these collections into programming syntax. This approach was adopted for the development of a query language with the aim of investigating the theoretical foundations for querying databases based on bags. The expressive power and complexity of this language, which can be regarded as a rational reconstruction of SQL, and its relationships with both relational algebra and a nested relational algebra [3] have been investigated.

The operations defined in this work are intended for *flat* bags (bags of tuples with attributes of basic types) as well as for *nested* bags (where tuple attribute values can also contain nested bags).

Types and domains. A distinction between types and domains is made. $[T_1, \ldots, T_n]$ is a tuple type, whose domain is the set of tuples over $T_1, \ldots, T_n$, that is, $\mathrm{Dom}([T_1, \ldots, T_n]) = \mathrm{Dom}(T_1) \times \ldots \times \mathrm{Dom}(T_n)$. $\{|T|\}$ is a bag type, whose domain is the set of finite bags of objects of type T.

Operations on records and tuples. The *tupling* function τ transforms a record into a tuple. Formally, $\tau(o_1, \ldots, o_k) = [o_1, \ldots, o_k]$, where $o_1, \ldots, o_k$ are k attributes of type $T_1, \ldots, T_k$, respectively, and $[o_1, \ldots, o_k]$ is a k-ary tuple, of type $[T_1, \ldots, T_k]$, containing o_i $(i = 1 \ldots k)$ in its i-th attribute.

The *attribute projection* function α_i returns the i-th attribute of a given tuple. Formally, $\alpha_i([o_1, \ldots, o_k]) = o_i$.

Membership test and multiplicity. An element n-belongs to a bag if it belongs to that bag and has exactly n occurrences. Function *member* of type $T \times \{|T|\} \rightarrow$ *Boolean* returns *true* on a pair (o, B) iff o p-belongs to B, $p > 0$.

In other words, in order to relate [26], [5], and Sect. 3.1, given a bag B and an element b,

1. b n-belongs to $B \Leftrightarrow b \in\in B = n \Leftrightarrow Occ(b, B) = n$;
2. $member(b, B) = true \Leftrightarrow b \in B$.

Constants. $\{|\cdot, \ldots, \cdot|\}$ is a bag; constant $\{||\}$ denotes the empty bag, even if also symbol $\emptyset$ is used very often with the same meaning.

Bagging (or bag singleton). Given an element o, by definition $\beta(o) = \{|o|\}$ is a bag containing o as a single element, i.e. o 1-belongs to $\beta(o)$.

Additive union. Bags are built by using the additive union operation $\uplus$ starting from the empty bag and singleton bags. That is, each bag is either $\{||\}$, or a singleton $\{|x|\}$, or the additive union of two bags $B \uplus B'$. Given two bags B and B' of type $\{|T|\}$, by definition $B \uplus B'$ is a bag of type $\{|T|\}$ such that o n-belongs to $B \uplus B'$ iff o p-belongs to B and q-belongs to B' and $n = p + q$.

This operator is the same as the concatenation operator $\sqcup$ defined by [5], here extended also to nested bags.

Extension. If f is a function of type $T \to \{|T'|\}$, the extension of f extends f to a function of type $\{|T|\} \to \{|T'|\}$ as follows:

$$EXT_f(\{|x_1, \ldots, x_n|\} \stackrel{\text{def}}{=} f(x_1) \uplus \ldots \uplus f(x_n) \ . \tag{112}$$

MAP_g is a concise form for $EXT_{\beta \circ g}$, where $\beta \circ g$ is the composition of function β (bagging) and function g, that is, $(\beta \circ g)(d) = \beta(g(d))$.

Cartesian product. If B and B' are bags containing tuples of arity k and k', respectively, by definition $B \times B'$ is a bag containing tuples of arity $k + k'$ such that $o = [a_1, \ldots, a_k, a_{k+1}, \ldots, a_{k+k'}]$ n-belongs to $B \times B'$ iff $o_1 = [a_1, \ldots, a_k]$ p-belongs to B, $o_2 = [a_{k+1}, \ldots, a_{k+k'}]$ q-belongs to B' and $n = p \cdot q$.

The Cartesian product operator is the same as the product for multisets defined in Sect. 3.1, which can be expressed in SQL by means of the select-from-where paradigm. Such an operator was naturally extended to nested multisets in Sect. 4.1.

Subtraction. Given two bags B and B' of type $\{|T|\}$, by definition $B - B'$ is a bag of type $\{|T|\}$ such that o n-belongs to $B - B'$ iff o p-belongs to B and q-belongs to B' and $n = \max(0, p - q)$.

This operator, called also *difference* [34], is the same as the difference operator $\setminus$ defined by [5] and outlined in Sect. 3.1. Such an operator was naturally extended to nested multisets in Sect. 4.1.

Maximal union. Given two bags B and B' of type $\{|T|\}$, by definition $B \cup B'$ is a bag of type $\{|T|\}$ such that o n-belongs to $B \cup B'$ iff o p-belongs to B and q-belongs to B' and $n = \max(p, q)$.

This operator is the same as the union operator $\cup$ defined by [5], here extended also to nested bags.

Intersection. Given two bags B and B' of type $\{|T|\}$, by definition $B \cap B'$ is a bag of type $\{|T|\}$ such that o n-belongs to $B \cap B'$ iff o p-belongs to B and q-belongs to B' and $n = \min(p, q)$.

This operator, also called *minimum intersection* [34], is the same as the intersection operator $\cap$ defined by [5] and outlined in Sect. 3.1. Such an operator was naturally extended to nested multisets in Sect. 4.1.

Duplicate elimination. Given a bag B of type $\{|T|\}$, $\epsilon(B)$ is a bag of type $\{|T|\}$ containing exactly one occurrence of each object of B. Formally, an object o 1-belongs to $\epsilon(B)$ iff o p-belongs to B for some $p > 0$, and 0-belongs to $\epsilon(B)$ otherwise.

Function ϵ, also called *unique* [34], becomes the identity function when it is turned from an operation on bags into its set analog.

Equality test. Function *eq* of type $T \times T \rightarrow Boolean$ returns *true* on a pair (o, o') iff o and o' are equal objects.

Subbag test. Function *subbag* of type $\{|T|\} \times \{|T|\} \rightarrow Boolean$ returns *true* on a pair (B, B') iff whenever o p-belongs to B, then o p'-belongs to B' for some $p' \geq p$.

In other words, relating [26] and [5], $subbag(B, B') = true \Leftrightarrow B \subseteq B'$. However, here the subbag test is inherent also to nested bags.

Complexity and expressive power. Complexity and expressive power are considered with respect to the language defined by operators τ, α_i, β, $\uplus$, EXT, $\times$, together with the empty bag constant. From the complexity point of view, each of the subtraction, union, intersection, duplicate elimination, subbag, membership, and equality test operators has polynomial time complexity with respect to the size of the input [34]. From the expressive power point of view, the same operators are related as follows [26, 34]:

1. $-$ can express all primitives other than ϵ;
2. ϵ is independent of the rest of the primitives;
3. $\cap$ is equivalent to *subbag* and can express both $\cup$ and *eq*;
4. *member* and *eq* are interdefinable, both are independent of $\cup$, and each of them, together with $\cup$, can express $\cap$.

Thus, the strongest combination of primitives among $-$, $\cup$, $\cap$, ϵ, *subbag*, *member* and *eq*, is $-$ and ϵ.

Bag query languages. Operators τ, α_i, β, $\uplus$, EXT, $\times$, $-$ and ϵ, together with the empty bag constant, are the primitives of *BALG (standard bag algebra)* [26], also referred to as *BQL (bag query language)* in [34].

BALG can express many operations commonly found in database languages. For instance, $MAP_{\lambda x.[\alpha_2(x),\alpha_3(x)]}$ denotes the projection of a tuple type (the type of the elements of the bag which is the operand) on its second and third arguments. Following [26], for the sake of brevity, the map projecting the attributes $i_1, \ldots, i_n$ will below be denoted by $\pi_{i_1,\ldots,i_n}$.

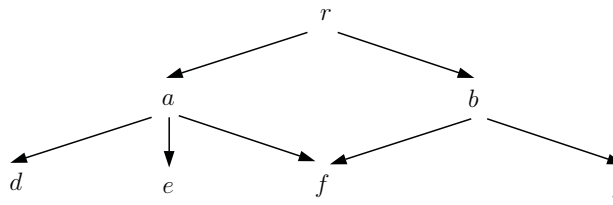

Fig. 4. The directed graph of Example 39

Likewise, $MAP_{\lambda x.\alpha_i(x)=a}$ denotes the selection within a bag of tuples of all and only the tuples wherein the i-th argument equals a. A shorthand for this query is $\sigma_{i=a}$.

Assumed to represent an integer i by means of a bag containing i occurrences of the same element (and nothing else), BALG allows the definition of several fundamental database primitives. For example, bags can be used to simulate SQL aggregate functions, such as `SUM` and `COUNT`. In fact, if B is a bag of tuples, $\texttt{COUNT}(B) = \pi_1(\{|a|\} \times B)$.

Example 38. A relation can be represented in BALG by means of a bag with no duplicates, such as $R = \{|7, 5, 3, 9, 4, 2|\}$. The *parity* of the cardinality of a relation becomes definable in BALG in the presence of an *order* on the domain. The following Boolean expression is *true* iff the parity of the cardinality of relation R is even:

$$\sigma_{\lambda x.\left(MAP_{[a]}(\sigma_{\lambda y.(y\leq x)}R)=(MAP_{[a]}(\sigma_{\lambda y.(x<y)}R)\right)}(R) \neq \{||\} \ . \tag{113}$$

In fact, the inequality holds only if there exists an x such that the number of elements smaller than or equal to x equals the number of elements strictly bigger than x. For instance, the above query can check the parity of the sample relation R by exploiting the order of integers. Based on Table 29, it emerges that the result of the left-hand side is $\{|4|\}$, thus the query gives *true*.

This example is quite simple as it involves a relation among integers. However, the same considerations hold also for relations among tuples, provided that there is at least one attribute whose domain is ordered. □

Example 39. Consider a directed graph whose edges are recorded in a binary relation G, such as the one displayed in Table 30 inherent to the graph of Fig. 4.

Formally, in BALG a graph is an object of type $\{|t \times t|\}$, where t is a base type with a countably infinite domain of uninterpreted constants. For instance, G is represented by the bag of tuples $\{|[r, a], [r, b], [a, d], [a, e], [a, f], [b, f], [b, g]|\}$.

Note that the domain of a graph is unordered, since the domain of t is unordered, that is, only equality test is available for constants of type t.

The following is a Boolean query on the graph, that is, a query defined on $\{|t \times t|\} \rightarrow Boolean$,

$$(\pi_2(\sigma_{2=a}(G))) - (\pi_1(\sigma_{1=a}(G))) \neq \{||\} \ . \tag{114}$$

Table 29. Parity of the cardinality of an ordered relation represented as a bag.

x	$\sigma_{\lambda y.(y \leq x)} R$	$MAP_{[a]}(\sigma_{\lambda y.(y \leq x)} R)$	$\sigma_{\lambda y.(x < y)} R$	$MAP_{[a]}(\sigma_{\lambda y.(x < y)} R)$
7	$\{\|7,5,3,3,4,2\|\}$	$\{\|[a],[a],[a],[a],[a]\|\}$	$\{\|9\|\}$	$\{\|[a]\|\}$
5	$\{\|5,3,4,2\|\}$	$\{\|[a],[a],[a],[a]\|\}$	$\{\|7,9\|\}$	$\{\|[a],[a]\|\}$
3	$\{\|3,2\|\}$	$\{\|[a],[a]\|\}$	$\{\|7,5,9,4\|\}$	$\{\|[a],[a],[a],[a]\|\}$
9	$\{\|7,5,3,9,4,2\|\}$	$\{\|[a],[a],[a],[a],[a],[a]\|\}$	$\{\|\|\}$	$\{\|\|\}$
4	$\{\|3,4,2\|\}$	$\{\|[a],[a],[a]\|\}$	$\{\|7,5,9\|\}$	$\{\|[a],[a],[a]\|\}$
2	$\{\|2\|\}$	$\{\|[a]\|\}$	$\{\|7,5,3,9,4\|\}$	$\{\|[a],[a],[a],[a],[a]\|\}$

It gives *true* iff the in–degree of a node a is bigger than its out–degree.

In our sample case, the following equalities hold: $\pi_2(\sigma_{2=a}(G)) = \pi_2(\{|[r,a]|\}) = \{|[a]|\}$, and $\pi_1(\sigma_{1=a}(G)) = \pi_1(\{|[a,e],[a,f]\}) = \{|[a],[a]|\}$. Thus, Equation (114) becomes $\{|[a]|\} - \{|[a],[a]|\} \neq \{||\}$, which gives *false*. □

BALG vs. relational algebra: complexity. As stated in [4], relational algebra enjoys an AC0 data complexity upper-bound and so does nested relational algebra [48]. AC0 [24] offers potential for efficient parallel evaluation. However, BALG is not in AC0: its authors state that BALG is contained in LOGSPACE.

BALG vs. relational algebra: expressiveness. A comparison of the expressive power of BALG to that of relational algebra is provided in [26]. Since BALG considers complex (nested) objects as well as flat relations, the relationship between BALG, when applied to complex objects, and nested relational algebra is also investigated.

Let *arithmetic* include the type *nat* of natural numbers, together with the operations of addition, multiplication, modified subtraction $\dot{-}$ (i.e. $n \dot{-} m = \max(0, n-m)$), and summation Σ_f. Here f is of type $T \to nat$, and Σ_f of type $\{T\} \to nat$ with the semantics $\Sigma_f(\{x_1, \ldots, x_k\}) = f(x_1) + \ldots + f(x_k)$.

A theorem states that BALG when restricted to flat bags is equivalent to relational algebra + *arithmetic*, and BALG over nested bags is equivalent to nested relational algebra + *arithmetic*. The comparison between BALG and nested relational algebra is restricted just to nested bags since it was shown [50] that nested relational algebra has no extra power with respect to relational algebra over flat relations in the sense that any nested relational algebra query from flat relations to flat relations can be defined in relational algebra.

Basically, BALG is more powerful than relational algebra and nested relational algebra since bags give a counting power. For instance, the result presented in Example 39 is quite important, as, while BALG can test the parity of relations over ordered domains, while relational algebra cannot test parity on any (ordered or unordered) domain.

However, some fundamental limitations on the expressive power of first-order logic, which are typical of relational algebra and nested relational algebra, still hold in BALG. For instance, consider the query *bag-even*, which tests the parity of the number of duplicates in a bag. Formally, given a bag B of type $\{|T|\}$ and an element c belonging to Dom(T), *bag-even(B,c) = true* if c's multiplicity in B

Table 30. The binary relation G representing the directed graph of Fig. 4.

Arrow tail node	*Arrow head node*
r	a
r	b
a	d
a	e
a	f
b	f
b	g

is even, and *false* otherwise. A proposition, which is claimed to be valid for both flat and nested bags, is stated in [26], according to which the query *bag-even* is not expressible in BALG. More generally, it is stated that a property of the number of duplicates of a single constant can be tested in BALG only if such a property (or its complement) holds for a finite number of bags.

Besides, a theorem states that, for any two graphs G and G', each of which consists of a number of disconnected simple cycles having the same length, there exists a k such that, if the lengths of the cycles of both graphs are longer than k, then, for any Boolean query q, $q(G) = q(G')$. This means that the two above defined graphs cannot be distinguished based on the answers to BALG queries. From this, [26] concludes that many recursive queries are not definable in BALG over unordered domains: parity of the cardinality of a relation, transitive closure, deterministic transitive closure, testing acyclicity, testing connectivity, testing for balanced binary trees. A proof of the undefinability in BALG of parity test, transitive closure, and test for balanced binary trees is provided in [34].

Several attempts to extend the power of BALG to support the above undefinable queries have to be registered. In this respect, three further operators, introduced in [33] and listed below, have to be considered.

Powerbag. Given a bag B, the *powerbag* operator returns the bag of all subbags of B. For instance,

$$powerbag(\{|1,1,2|\}) = \{|\ \{||\}, \{|1|\}, \{|1|\}, \{|2|\}, \{|1,1|\}, \{|1,2|\}, \{|1,2|\}, \{|1,1,2|\}\ \}\ . \quad (115)$$

Powerset. When applied to a bag B, the *powerset* operator returns the bag of all subbags of B, each with multiplicity 1. For instance,

$$powerset(\{|1,1,2|\}) = \{|\{||\}, \{|1|\}, \{|2|\}, \{|1,1|\}, \{|1,2|\}, \{|1,1,2|\}\}\ . \quad (116)$$

This operator is the same as the homonymous operator defined in [5]; however, here it is extended to nested bags.

Loop. Construct *loop* takes as input a function f of type $T \to T$, an object o of type T, and a bag $B = \{|o_1, \ldots, o_n|\}$ of any type $\{|T'|\}$, and returns f applied n times in cascade to o (where n is the cardinality of B).

Power and complexity of extended BALG. It was shown in [33] that *powerbag* and *powerset* are interdefinable. However, even if all the queries which are not definable in BALG over unordered domains become definable in BALG+*powerset*, they are not definable efficiently. Out of efficiency BALG+*powerbag* has to be preferred. The complexity of BALG+*powerset* and BALG+*powerbag* was studied in [27].

The *loop* operator was introduced as an alternative to the structural recursion operator. The main problem with using structural recursion is that it requires some preconditions on its parameters for well-definedness. However, such preconditions are generally undecidable [9]. In [33] it was shown that nested BALG with structural recursion is equivalent to nested BALG with *loop*.

7 Conclusion

This paper has shown how the notion of a multiset affects the data model of database systems and the relevant query languages. Since most current database systems are relational, the focus is on the relational data model and on relational algebra.

The relational data model is based on the central concept of a relation. Intuitively, a relation is a set of tuples, where each tuple is, in turn, a collection of atomic values, one for each attribute of the relation. A database consists of one or several relations. Once a relational database has been created, it is possible to retrieve and modify the stored information by means of appropriate query languages. Among formal query languages for relational databases is relational algebra, which deals with the manipulation of sets. This means that each query result is expected not to include duplicates.

Even if, from the theoretical point of view of relational algebra, the result of a query is a set, from the practical point of view it is sometimes convenient to have multisets as results of queries, in particular if the information they include is intended to be processed by aggregate functions. Thus, practical query languages for relational databases, typically based on the standard query language SQL, allow the manipulation of relations in a multiset-oriented way. Based on the above considerations, relational algebra operators have been straightforwardly extended in order to manage multisets, thus obtaining some relational algebras for multisets.

Historically, the awareness has gradually grown that the relational data model is too simple to express in an intuitive way complex data, as those typically encountered in non business applications, and relationships among such data. Thus, several extensions to the relational model have been proposed as well as a number of extended relational algebras. The most noteworthy extension of the relational model, called nested relational model, was the transformation of usual (flat) relations into nested relations, where each attribute value is not necessarily atomic, rather, it can be a (nested) relation itself. The algebraic counterpart of the nested relational model is a nested relational algebra, which,

although manipulating only sets, both extends relational algebra operators to nested relations and introduces new operators.

Roughly, two research streams have been focused on the extension of relational algebra. On the one hand, flat relations have been extended to flat multisets. On the other, flat relations have been extended to nested relations. The two approaches have been integrated within the notion of a complex relation, that is, a multiset of tuples where each attribute is possibly a (complex) relation. A nested relational algebra for complex relations has been defined by combining the concepts inherent to the relational algebra for multisets and the nested relational algebra.

Moreover, the paper hints at some research topics inherent to multisets in database systems. In the last decade, there has been some activity in trying to identify a standard formal query language for multisets, which may play a role similar to that of relational algebra. The first requirement of such a language is to support aggregate functions, and the second is to deal with nested data. Processing and optimization of queries in the proposed languages is an active research area. However, investigating the expressive power of multiset languages involves many difficulties. In fact, in order to cope with aggregate functions, multiset languages support built-in arithmetic and, therefore, it is hard to find a logic that captures them and that can be exploited for proving results about the expressive power. As to nested multiset languages, they involve calculi that are essentially higher-order logics, whose expressibility properties are rather unknown.

References

1. S. Abiteboul and N. Bidoit. Non-first-normal form relations to represent hierarchically organized data. In *Third ACM SIGMOD SIGACT Symposium on Principles of Database Systems*, 1984.
2. S. Abiteboul and N. Bidoit. Nonfirst normal form relations: An algebra allowing data restructuring. *Journal of Comp. and System Sc.*, 33(1):361–393, 1986.
3. S. Abiteboul, P.C. Fisher, and H.J. Schek, editors. *Nested Relations and Complex Objects in Databases.* Number 361 in LNCS. Springer-Verlag, Berlin, Germany, 1989.
4. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, Reading, MA, 1994.
5. J. Albert. Algebraic properties of bag data types. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Seventeenth International Conference on Very Large Data Bases*, pages 211–219, Barcelona, Spain, 1991.
6. H. Arisawa, K. Moriya, and T. Miura. Operation and the properties on non-first-normal-form relational databases. In *Ninth International Conference on Very Large data Bases*, pages 197–204, Florence, Italy, 1983.
7. S. S. Bhowmick, S. K. Madria, W. K. Ng, and E.P. Lim. Web bags – are they useful in a web warehouse? In *Fifth International Conference on Foundations of Data Organization*, Kobe, Japan, 1998.
8. N. Bidoit. The Verso algebra or how to answer queries with fewer joins. *Journal of Computer and System Sciences*, 35(3):321–364, 1987.

9. V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *Eighteenth International Colloquium on Automata, Languages, and Programming*, number 510 in LNCS, Madrid, Spain, 1991.
10. F. Cacace and G. Lamperti. *Advanced Relational Programming*, volume 371 of *Mathematics and Its Applications*. Kluwer Academic Publisher, Dordrecht, The Netherlands, 1997.
11. L. Cardelli. Types for data-oriented languages. In J. W. Schmidt, S. Ceri, and M. Missikoff, editors, *International Conference on Extending Database Technology*, number 303 in LNCS, pages 1–15, Venice, Italy, 1988.
12. M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A data model and query language for EXODUS. In H. Boral and P. Larson, editors, *ACM SIGMOD International Conference of Management of Data*, pages 413–423, Chicago, IL, 1988.
13. J. Celko. *SQL for Smarties*. Morgan-Kaufmann, San Francisco, CA, 1995.
14. S. Ceri, S. Crespi-Reghizzi, G. Lamperti, L. Lavazza, and R. Zicari. Algres: An advanced database system for complex applications. *IEEE-Software*, 7(4):68–78, 1990.
15. D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM Journal of Research and Development*, 20(6):560–575, 1976.
16. E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
17. L.S. Colby. A recursive algebra and query optimization for nested relations. In *ACM SIGMOD Conference on Management of Data*, pages 273–283, Portland, OR, 1989.
18. P. Dadam, K. Küspert, F. Andersen, H. Blanken, R. Erbe, J. Günauer, V. Lum, P. Pistor, and G. Walch. A DBMS prototype to support extended NF2 relations: An integrated view on flat tables and hierarchies. In *ACM SIGMOD Conference on Management of Data*, pages 356–366, Washington, DC, 1986.
19. C. J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, Reading, MA, 1993.
20. U. Dayal, N. Goodman, and R. H. Katz. An extended relational algebra with control over duplicate elimination. In *ACM Symposium on Principles of Database Systems*, pages 117–123, Los Angeles, CA, 1982.
21. V. Desphande and P.Å. Larson. An algebra for nested relations. Technical Report Research Report CS-87-65, University of Waterloo, Waterloo, Ontario, December 1987.
22. P.C. Fischer and S.J. Thomas. Operators for non-first-normal-form relations. In *Seventh International Computer Software Applications Conference*, pages 464–475, Chicago, IL, 1983.
23. D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrette, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M.C. Shan. Iris: An object-oriented database management system. *ACM Transactions on Information Systems*, 5(1):48–69, 1987.
24. M. Furst, J. B. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical System Theory*, 17:13–27, 1984.
25. T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In M. J. Carey and D. A. Schneider, editors, *ACM SIGMOD International Conference on Management of Data*, pages 328–339, San Jose, CA, 1995.

26. S. Grumbach, L. Libkin, T. Milo, and L. Wong. Query languages for bags: Expressive power and complexity. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 27, 1996.
27. S. Grumbach and T. Milo. Towards tractable algebras for bags. In *Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 49–58, Washington, DC, 1993.
28. M. Hammer and D. McLeod. Database description with SDM: A semantic database model. *ACM Transaction on Database Systems*, 6(3):351–386, 1981.
29. G. Jaenschke. Recursive algebra for relations with relation valued attributes. Technical Report 85.03.002, Heidelberg Scientific Centre, IBM, Heidelberg, Germany, 1985.
30. G. Jaenschke and H.J. Schek. Remarks on the algebra for non first normal form relations. In *First ACM SIGACT SIGMOD Symposium on Principles of Database Systems*, pages 124–138, 1982.
31. A. Klausner and N. Goodman. Multirelations - semantics and languages. In A. Pirotte and Y. Vassiliou, editors, *Eleventh International Conference on Very Large Data Bases*, pages 251–258, Stockholm, Sweden, 1985.
32. A. Levy. Answering queries using views: A survey, 2000. http://www.cs.washington.edu/homes/alon/site/files/view-survey.ps.
33. L. Libkin and L. Wong. Some properties of query languages for bags. In C. Beeri, A. Ohori, and D. Shasha, editors, *Fourth International Workshop on Database Programming Languages – Object Models and Languages*, Workshops in Computing, pages 97–114, New York, NY, 1993.
34. L. Libkin and L. Wong. Query languages for bags and aggregate functions. *Journal of Computer and System Sciences*, 55(2):241–272, 1997.
35. S. K. Madria, S. S. Bhowmick, W. K. Ng, and E. P. Lim. Research issues in web data mining. In M. K. Mohania and A. Min Tjoa, editors, *Data Warehousing and Knowledge Discovery International Conference*, pages 303–312, Florence, Italy, 1999.
36. F. Manola and U. Dayal. PDM: An object-oriented data model. In K. R. Dittrich and U. Dayal, editors, *International Workshop on Object-Oriented Database Systems*, pages 18–25, Pacific Grove, CA, 1986.
37. J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide.* Morgan-Kaufmann, San Francisco, CA, 1993.
38. I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Sixteenth International Conference on Very Large Data Bases*, pages 264–277, Brisbane, Australia, 1990.
39. G. Ozsoyoglu, Z.M. Ozsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12(4):566–592, 1987.
40. M.A. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for ¬1NF relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, 1988.
41. H.J. Scheck and M.H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.
42. H.J. Schek and P. Pistor. Data structures for an integrated database management and information retrieval systems. In *Eighth International Conference on Very Large Data Bases*, pages 197–207, Mexico City, Mexico, 1982.

43. M.H. Scholl, S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, and A. Verroust. Verso: A database machine based on nested relations. In S. Abiteboul, P.C. Fischer, and H.J. Schek, editors, *Nested relations and complex objects in databases*, number 361 in LNCS. Springer-Verlag, Heidelberg, Germany, 1989.
44. P. M. Schwarz, W. Chang, J. C. Freytag, G. M. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the Starburst database system. In K. R. Dittrich and U. Dayal, editors, *roceedings of the International Workshop on Object-Oriented Database Systems*, pages 85–92, Pacific Grove, CA, 1986.
45. D. W. Shipman. The functional data model and the data language DAPLEX. *ACM Transaction on Database Systems*, 6(1):140–173, 1981.
46. D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Twenty-second International Conference on Very Large Data Bases*, pages 318–329, Mumbai, India, 1996.
47. M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Transaction on Database Systems*, 1(3):189–222, 1976.
48. D. Suciu and V. Tannen. A query language for NC. In *Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Minneapolis, Minnesota, 1994.
49. S. L. Vandenberg and D. J. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritance. In J. Clifford and R. King, editors, *ACM SIGMOD International Conference on Management of Data*, pages 158–167, Denver, CO, 1991.
50. L. Wong. Normal form and conservative properties for query languages. In *Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Washington, DC, 1993.