ISSN 0081-5438, Proceedings of the Steklov Institute of Mathematics, 2011, Vol. 274, pp. 273–288. © Pleiades Publishing, Ltd., 2011.
Original Russian Text © M. A. Taitslin, 2011, published in Trudy Matematicheskogo Instituta imeni V.A. Steklova, 2011, Vol. 274, pp. 297–313.

Comparison of Expressive Power of Some Query Languages for Databases

M. A. Taitslin [a]

Received March 2011

To outstanding mathematician and great person
Sergei Ivanovich Adian on the occasion of his 80th birthday

Abstract—A version of the SQL language and a version of the stratified Datalog are considered, and it is proved that each of these languages can be translated into the other.

DOI: 10.1134/S0081543811060174

A decisive role in my scientific career was played by my teacher, the outstanding Russian mathematician Anatolii Ivanovich Mal'tsev. After his decease, I faced great difficulties that I could have hardly overcome without constant support from Sergei Ivanovich Adian. I consider it a great honor to take part in this collection of scientific papers.

1. INTRODUCTION

After the famous studies by Codd (see [9, 10]), the relational model, in which a database is considered as a finite set of finite tables, has become a generally accepted database model. Each table has a predetermined number of columns with fixed names. The name of a table itself is also fixed in advance. Only the number of rows and the rows themselves may change.

There exist several languages that are convenient to extract information on the data stored in a database. The most popular (and, possibly, the only one as of late) language is the SQL language. The description of the SQL standards and the methods of programming in SQL can be found in many manuals, for example, in [18, 20–22].

It was noticed back by Codd that SQL is in fact a modification of the language of the predicate logic. The main drawback of the old SQL standards is that there exist very easily recognizable queries that could not be defined in the SQL. For example, the connectedness of a graph cannot be defined by a formula of the predicate logic. This fact was pointed out back in [8]. Therefore, the later SQL standards involve recursive queries.

Another well-known language is Prolog. However, it was noticed long ago that, possibly, the strategy of bottom-up evaluation is more efficient compared with the strategy of Prolog; see, for example, [13]. This is how the Datalog language appeared. As an independent section of logic programming, Datalog has been developed since 1977 after the well-known symposium on logic and databases (see [1]).

When negations are admitted in the rules, the meaning of the latter becomes ambiguous. A variety of interpretations have been proposed for negations; however, none of them are satisfactory enough. Therefore, a stratified Datalog has been developed, in which negations are admitted, but only of those relations whose evaluation is already completed.

[a] Chair of Informatics, Faculty of Applied Mathematics and Cybernetics, Tver State University, Sadovyi per. 35, Tver, 170004 Russia.

E-mail address: Michael.Taitslin@tversu.ru

When defining the elements of tables, one faces the problem of where these elements are taken from. It is convenient to fix in advance a domain from which the elements of tables are taken. But then it immediately becomes possible to use relations defined on this universe in the formulation of queries. This has already been admitted in the first standards of the SQL. In this situation, a lot depends on the choice of the universe.

Here we take the set of natural numbers as the universe. If we make such a choice, then the answers to queries defined even by formulas of the predicate logic may not be finite tables. For example, the query $x > 2$ has an infinite set as an answer. Kanellakis et al. (see [14]) suggested considering finitely representable relations instead of finite tables. In this case, an answer to an SQL query that, say, does not contain recursive calls is again finitely representable. By finitely represented relations, these authors meant relations defined by quantifier-free formulas.

However, for a response to a finitely represented query to be finitely represented, it is convenient to consider, along with the ordinary linear order, the so-called gap-orders $<_p$ for all natural p. In this case, if one adds constants for all natural numbers and the relations $x =_p y$ for formulas $x + p = y$, each formula becomes equivalent to a positive quantifier-free formula.

The third long-studied language is the language of fixed point logic.

This logic has been used by many authors (see, for example, [2–5, 7, 11, 12, 15]). One of the main results here is the exact characterization of queries that are computable in polynomial time as queries defined by formulas of fixed point logic (the Livchak–Immerman–Vardi theorem). The distinctive feature of our approach is that we consider a structure that is not finite. Therefore, not every operator of a fixed point gives a result. In some cases, the process may never end.

In a sense, the mutual translatability of the formulas of fixed point logic and of stratified Datalog programs is also well known (see, for example, [15]). The specific feature of our consideration is that the object domain is infinite.

Here we prove that for a certain choice of definitions, all the three languages have the same expressive power.

2. DATALOG

We consider the set $\omega = \{0, 1, 2, \ldots\}$ of natural numbers together with the relation $<$ of usual linear order and with a distinguished element 0. Along with the relation $<$, we will also consider relations $<_p$ for fixed positive integers p. Here $x <_p y$ denotes that $(x + p) < y$; in other words, this means that there are p distinct numbers between x and y that are different from x and y. By $x <_0 y$ we just mean $x < y$. Let $x =_i y$ for a positive integer i be an abbreviation for $x <_{(i-1)} y \,\&\, \neg\, x <_i y$. Let $x =_0 y$ be $x = y$. Although 0 and these relations $<_p$ and $=_p$ are defined on this linearly ordered set of natural numbers by formulas of the predicate logic, we add them in order that each formula in the enrichment thus obtained be equivalent to a quantifier-free formula that does not contain negations. Here and below, by a formula we mean a formula of the predicate logic (see, for example, [19]) that does not contain implications. All the other terms and notations, which are not explained here, are also borrowed from [19].

Lemma 2.1. *In the system $(\omega, 0, \{=_p, <_p \mid p \in \omega\})$, each formula is equivalent to a quantifier-free formula that does not contain negations. In the system $(\omega, =, \{p, <_p \mid p \in \omega\})$, each existential formula that does not contain negations is equivalent to a quantifier-free formula that does not contain negations.*

Proof. As usual, it suffices to consider the case of a formula of the form $(\exists x)\Phi$, where Φ is a quantifier-free formula. It suffices to consider only the case when Φ is a conjunction of basic formulas (in other words, atomic formulas and negations of atomic formulas). However, the negation of the formula $x < y$ can be replaced by $y < x \vee y = x$, and the negation of the formula $x <_p y$ for $p > 0$

can be replaced by

$$y < x \vee y = x \vee x =_1 y \vee \ldots \vee x =_{(p-1)} y.$$

There exist a finite number of different possibilities for the mutual disposition of the free variables encountered, x, and 0. If x falls into the interval between y and a free variable z (in this case, y may be either a free variable or 0), then it suffices to remove the quantifier with respect to x, appropriately correcting the distance between y and z. $\square$

Formulas that do not contain negations are said to be positive.

Let $\Sigma = \langle 0, \{=_p, <_p \mid p \in \omega\}\rangle$.

Atomic formulas of signature $\Sigma_0 = \langle =, \{p, <_p \mid p \in \omega\}\rangle$ are called *atoms*.

In what follows, we consider a finite set of symbols of relations with the number of argument places (the arity) of each symbol specified. These symbols are called *input* or *external* symbols, by which we mean that the interpretations of these symbols represent the information stored. In other words, we are given a *database*. The above-mentioned finite set of symbols of relations with the arity of each relation specified is called a *scheme of this database*.

To write a stratified Datalog program, one needs another finite set of symbols of relations that does not contain input symbols. These symbols of relations are called *internal* symbols or internal names. For each internal symbol, the number of argument places is also specified. Sometimes internal symbols are divided into *output* symbols, which contain the result of a program, and *intermediate* symbols, which serve for storing intermediate results. All internal symbols are split into ranks. The ranks start from 1 and include several consecutive natural numbers. There always exist internal symbols of rank 1. Internal symbols of rank $(i + 1)$ exist only if there exist internal symbols of rank i. There also exists an infinite set of object variables. As usual, object variables can be interpreted as words in an alphabet, for example, consisting of $|$ and x. For example, x is a variable, and if y is a variable, then $y|$ is also a variable. However, naturally, a specific method of representing variables is not essential.

The meaning of the ranking is that we consider a stratified negation. One can use the negation of an internal name, but only after completing the calculation of the value of this name.

The formal definition is as follows. A stratified Datalog program is a finite sequence of rules. Each rule consists of a head and a body.

The head of a rule has the form $P(x_1, \ldots, x_n)$, where P is an internal n-ary symbol and $x_1, \ldots, x_n$ is a sequence of pairwise different variables. The rank of a rule is the rank of the symbol P. We refer to P as the name of the rule.

The body of a rule is either empty or consists of a finite sequence of formulas. Each formula in the body of a rule has one of the following forms:

- an atom;
- $Q(y_1, \ldots, y_k)$, where Q is an external k-ary symbol and $y_1, \ldots, y_k$ is a sequence of variables;
- $Q(y_1, \ldots, y_k)$, where Q is an internal k-ary symbol whose rank is not greater than the rank of the rule, and $y_1, \ldots, y_k$ is a sequence of variables;
- $\neg Q(y_1, \ldots, y_k)$, where Q is an internal k-ary symbol whose rank is *strictly less* than the rank of the rule, and $y_1, \ldots, y_k$ is a sequence of variables.

Usually, the head of a rule is separated from the body by the sign $\leftarrow$.

It is not required that each variable entering the body of a rule should enter its head. Let us consider a tuple of values of the variables of the head of a rule. The meaning of a rule is that if all formulas of the body become true for some values of the variables that do not enter the head of the rule, then this tuple of values of the variables of the head is included in the interpretation of the name of the rule. The formal definition of the operation of a stratified Datalog program is as follows.

By a *finite representation* we mean assigning a positive quantifier-free formula of signature Σ_0 to each external symbol of a relation, provided that the number of different variables contained in this formula is not more than the arity of this symbol. We assume that the variables are ordered (for example, by the number of | after the first letter if the variables are given by words starting with this letter followed by the symbols |). Adding, when necessary, conjunctive terms of the form $z = z$, we can assume that the number of different variables in the assigned formula is equal to the arity of the external symbol under consideration. For definiteness, we assume that the first variable gives the value of the first coordinate of the relation, the second variable, the second coordinate, etc., and the last variable, the last coordinate. Note that each finite table can certainly be finitely represented. Indeed, the tuple $(i_1, \ldots, i_k)$ is represented by the formula $(x_1 = i_1 \,\&\, \ldots \,\&\, x_k = i_k)$. A table of a finite number of rows can be represented by a disjunction of such formulas.

We consider a linear order on the variables, assuming that the variables are numbered by natural numbers and that this order is induced by the order on the numbers of variables. We denote this order relation by the symbol $<$. It is clear that this does not lead to confusion with the order on the values of these variables.

Here and below in similar situations, we assume that $x_1 < \ldots < x_k$. This means that the number of x_1 is always less than the number of x_2, the number of x_2 is less than the number of x_3, etc., and the number of x_{k-1} is less than the number of x_k. The expression $Q(y_1, \ldots, y_k)$ implies that the arity of Q is equal to k, that $y_1 < \ldots < y_k$ in the order on the variables, and that the list $y_1, \ldots, y_k$ is the list of all variables entering the formula $Q(y_1, \ldots, y_k)$. The formula $Q(y_1, \ldots, y_k)$ is true for a given tuple $(i_1, \ldots, i_k)$ of natural numbers if this formula is true when y_1 takes the value i_1, $\ldots$, y_k takes the value i_k.

A *state of a database* is a certain finite representation of this database. A state σ of a database assigns a formula $\sigma(R)$ to each external symbol R. The initial state assigns empty sets of tuples of natural numbers to all internal symbols; the external symbols are assigned formulas that do not change during the operation of the program. The sets assigned to internal symbols increase according to the following rules.

The program runs step by step. The steps are numbered by pairs (i, j) of positive integers. The first step is numbered by the pair $(1, 1)$. At step (i, j), the sets assigned to the internal symbols of rank i are changed. The pairs (i, j) are ordered lexicographically.

For a given tuple $(i_1, \ldots, i_k)$ of natural numbers and a k-ary symbol Q, the formula $Q(y_1, \ldots, y_k)$ is said to be true if this tuple is included at this step in the set of tuples assigned to this symbol. Any external symbol is assigned the set of tuples on which the formula assigned to this symbol is true. For a given tuple $(i_1, \ldots, i_k)$ of natural numbers and a k-ary symbol Q, the formula $\neg Q(y_1, \ldots, y_k)$ is said to be true at step (i, j) if Q is an internal symbol whose rank is less than i and this tuple was not included at the previous steps in the set of tuples assigned to this symbol.

If it turns out at step (i, j) that there are no internal symbols of rank i, then the program terminates. The result of the program consists of the sets assigned to internal symbols. Otherwise only the sets assigned to internal symbols of rank i may change at step (i, j).

Let P be an internal n-ary symbol of rank i. A tuple $(j_1, \ldots, j_n)$ is included at step (i, j) in the set assigned to the symbol P in the following cases:

- if this tuple was included in the set assigned to this symbol earlier;
- if the program contains a rule

$$P(x_1, \ldots, x_n) \leftarrow \Phi_1 \ldots \Phi_m$$

and one can choose natural numbers as the values of variables that do not enter the head of this rule in such a way that, after assigning x_1 the value j_1, $\ldots$, x_n the value j_n, all the formulas $\Phi_1, \ldots, \Phi_m$ turn out to be true.

If a tuple $(j_1, \ldots, j_n)$ is included at step (i, j) in the set assigned to the symbol P and this tuple was not included in this set at the previous steps, then this tuple is added to this set. If no tuple is added to any set at step (i, j), then the program goes to step $(i + 1, 1)$. Otherwise the program goes to step $(i, j + 1)$.

Lemma 2.2. *After step* (i, j) (*before executing the next step*), *every set assigned to an internal symbol can be defined by a quantifier-free positive formula of signature* Σ. *Moreover, after step* $(1, i)$, *every set assigned to an internal symbol can be defined by a quantifier-free positive formula of signature* Σ_0.

Proof. The lemma is proved by induction. Before step $(1, 1)$, the assertion of the lemma obviously holds. If this assertion holds before a certain step, then it is also valid after this step. This follows from Lemma 2.1. The point is that each rule with name P changes only the formula assigned to this P. In this case, this formula is replaced by a disjunction of this formula and some other existential formula of signature Σ. □

Thus, if the program terminates, then the result of the program is a certain set of quantifier-free positive formulas of signature Σ. It is this fact that justifies the use of positive quantifier-free formulas instead of finite tables, because the result of a Datalog program for finite tables may not be a set of finite tables, but is always given by a set of positive quantifier-free formulas.

If all internal symbols have rank 1, then the program always terminates (see [6, Theorem 4.1]). Since we consider a somewhat different situation and the formulation and proof of Theorem 4.1 in [6] are presented not quite accurately, here we give a full proof.

A Datalog program is said to be *nonstratified* if either all internal symbols have rank 1 or the internal symbols are not assigned ranks at all. This term is generally accepted. If internal symbols are not assigned ranks at all, we will assume that all internal symbols are assigned rank 1. A Datalog program is said to be *safe* if it terminates for every state.

Theorem 2.3. *Every nonstratified program is safe.*

First, we recall two well-known facts whose proofs are given, for example, in [6] (see facts 4.3 and 4.4 there). We consider a partial order on finite sequences of natural numbers of the same length. Of two different sequences of this kind, we consider the first to be the smaller one and the second the larger one if each term of the first sequence is not greater than the corresponding term of the second sequence. By the corresponding terms we mean the first terms, the second terms, etc., and, finally, the last terms. In a given set of such sequences, a certain sequence is said to be *minimal* (*maximal*) if this set does not contain a sequence smaller (larger) than the sequence in question. Two such sequences are said to be comparable if one of them is smaller than the other.

Remark 2.4. In any set of finite sequences of natural numbers of the same length, the subset of minimal sequences is finite.

Remark 2.5. Let k be a positive integer. Any sequence of pairwise different finite sets of finite sequences of length k composed of natural numbers in which each sequence from each next set is either incomparable to any sequence from the preceding set or is smaller than or equal to some sequence from the preceding set is finite.

Proof of Theorem 2.3. The program employs a finite number of positive integers and a finite number of variables for each name. Therefore, for each internal name, there exist only a finite number of possible mutual dispositions of these variables and numbers. By a mutual disposition we mean the conjunction of all true formulas of the forms $x < y$ and $x = y$ in which each of x and y is either one of the numbers or one of the variables under consideration. Since there are only a finite number of possible mutual dispositions, it suffices to prove that for a fixed internal name and a fixed possible mutual disposition, there is a step of the program after which no tuples with this mutual disposition are added to the set assigned to this internal name. Since there are only a finite number

of possible values of each variable that do not exceed a certain number, for each mutual disposition we will study the sets of values only of those variables that are greater than all the numbers. For such a set, we will consider a tuple of distances between adjacent elements of the set and between the least element of the set and the greatest number. In greater detail, let n be the greatest of all natural numbers encountered in the program. Fix a certain possible mutual disposition of natural numbers encountered in the program and the variables of the internal name under study. Let $z_1, \dots, z_k$ be the set of all those variables in the order of increasing values that are greater than n. Since we have fixed the mutual disposition, this set is defined uniquely. With every tuple $(d_1, \dots, d_k)$ of values of $z_1, \dots, z_k$, we associate a tuple of natural numbers $(d_1 - n, d_2 - d_1, \dots, d_k - d_{k-1})$. We call this tuple the set of distances for the tuple $(d_1, \dots, d_k)$. We call $(d_1, \dots, d_k)$ a tuple *with minimal distances* among a certain set of such tuples if the set of distances for this tuple is minimal (this set does not contain another tuple whose set of distances is such that each of its coordinates is not greater than the corresponding coordinate in the set of distances for the tuple under consideration). Similarly, we say that a second tuple has greater distances than a first one if each coordinate in the set of distances for the first tuple is not greater than the corresponding coordinate in the set of distances for the second tuple.

Lemma 2.6. *If, at some computation step $(1, i)$, a tuple is included in the set of tuples assigned to a certain internal name, then at this step all the tuples with greater distances are included in the same set.*

Proof. Let us examine what goes on at step $(1, i)$. A tuple $(j_1, \dots, j_n)$ is added to the set assigned to some internal symbol P if there exists a rule

$$P(x_1, \dots, x_n) \leftarrow \Phi_1 \dots \Phi_m$$

and one can choose natural numbers as the values of the variables that do not appear in the head of this rule, in such a way that, after assigning x_1 the value j_1, $\dots$, x_n the value j_n, all the formulas $\Phi_1, \dots, \Phi_m$ turn out to be true. These formulas are either quantifier-free positive formulas of signature Σ_0 assigned to external symbols, or atoms, or quantifier-free positive formulas of signature Σ_0 assigned to internal symbols before executing the step $(1, i)$. As already mentioned, each rule with name P changes only the formula assigned to this P. In this case, the formula is replaced by the disjunction of this formula and a certain existential positive formula of signature Σ_0. As pointed out in Lemma 2.2, this disjunction is equivalent to a quantifier-free positive formula of signature Σ_0. It remains to notice that if a quantifier-free positive formula of signature Σ_0 is satisfied by some tuple of natural numbers, then this formula is also satisfied by any tuple with greater distances. But this is quite obvious. □

Therefore, it suffices to realize that no new tuples with minimal distances arise after a certain step of the program. This fact follows from the remarks made above. □

Theorem 2.3 pertains to nonstratified programs; however, a stratified program can simulate the operation of an arbitrary Turing machine and fail to terminate if this Turing machine does not stop (see [17] for more details).

A Datalog program can define a transitive closure of an input relation. For example, for an input binary symbol P and an internal binary symbol Q, the program may contain the rules

$$Q(x, y) \leftarrow P(x, y), \qquad Q(x, y) \leftarrow Q(x, z)\ P(z, y).$$

If P is arbitrary, then this program may not terminate; however, for a finitely represented P this program will always terminate. This fact shows that Theorem 2.3 is not quite trivial. In particular, it states that for a finitely represented binary relation the length of the minimal path between two vertices in the corresponding infinite graph always does not exceed a preset natural number, provided that there exists at least one path from the first vertex to the second.

The following program is possibly the simplest one that does not terminate for a finitely represented P:

$$Q(x) \leftarrow P(x), \qquad Q(x) \leftarrow Q(z)\ z =_2 x.$$

This program can be rewritten as a stratified program by assigning rank 2 to $Q(x)$ and adding a binary internal symbol Q_1 and the rules

$$Q_1(x,y) \leftarrow x <_2 y, \qquad Q(x) \leftarrow P(x), \qquad Q(x) \leftarrow Q(z)\ z <_1 x\ \neg Q_1(z,x).$$

If $P(x)$ is defined as $x = 2$, the program does not terminate. This shows that for the validity of the assertion that nonstratified programs always terminate, by atoms we should mean atomic formulas of signature Σ_0.

A similar example shows that for Theorem 2.3 to hold, one should require that external symbols be assigned quantifier-free positive formulas of signature Σ_0. However, if we do not care about the validity of Theorem 2.3, then external symbols can be assigned any formulas of signature Σ. This will not extend the class of relations resulting from Datalog programs. In other words, any formula of signature Σ can be obtained from a certain Datalog program with a set of quantifier-free positive formulas of signature Σ_0 as an input.

3. SQL

3.1. Simple cases. We propose a version of the SQL in which the elements of the rows in the tables are taken from the set of natural numbers and, in addition to the names of tables, only symbols of the signature $\Sigma = \langle 0, \{=_p, <_p \mid p \in \omega\}\rangle$ are used in queries. In this case, each natural number p can be represented as x subject to the condition $0 =_p x$. As already mentioned, each finite table is defined by some quantifier-free positive formula of signature Σ_0. However, as before, we will consider a more general case of an arbitrary quantifier-free positive formula of signature Σ_0, whose interpretation may not be a finite table.

In what follows, it is convenient to refer to quantifier-free positive formulas of signature Σ as tables, and to the tuples of values of the variables in such a formula for which the latter is true as the rows of this table.

So, we fix a finite set of symbols of relations with the number of argument places of each symbol specified. We will call these symbols *input* or *external* symbols, bearing in mind that their interpretations represent the stored information. In other words, we are given a database. Each external symbol is assigned a quantifier-free positive formula of signature Σ_0, and the number of variables in this formula coincides with the number of argument places of this symbol. As already mentioned, the variables are linearly ordered, and we assume for definiteness the first variable to be the value of the first coordinate of the relation, the second variable, the second coordinate, etc., and the last variable, the last coordinate. The assigned formulas may be arbitrary, but we assume that the sequence of variables is fixed for every external symbol. In other words, we assume that the names of columns in each table are preset. A variable is said to be assigned to an external symbol or included in this symbol if this variable appears in the formula assigned to this symbol.

An SQL program results in a finite set of formulas.

For example, given a table $R(x,y)$, we want to find the rows in which $x <_2 y$. This can be done by the following query:

```
select *
  from R
  where x <_2 y
```

As a result, we obtain a table that contains those and only those rows of the table R in which $x + 2 < y$.

A formal definition is as follows.[1] An SQL program is a finite sequence of queries. In the simplest case, a query is defined as

```
select <list of arguments>
  from <name of table>
```

Here ⟨name of table⟩ is one of external symbols; ⟨list of arguments⟩ is a sequence of variables each of which appears in the formula assigned to this external symbol. In a list of arguments, the arguments are separated by a comma.

If a list of arguments represents all arguments arranged in the same order, one can write $*$ instead of the list of arguments.

If we wish to impose a certain condition on the resulting rows, we need the instruction `where`:

```
select <list of arguments>
  from <name of table>
  where <condition>
```

As a condition, one can take any quantifier-free positive formula of signature Σ such that the variables in this formula are encountered among the variables of the formula assigned to the external symbol ⟨name of table⟩.

In a more complex case, a query is given by

```
select <list of arguments>
  from <list of tables>
  where <condition>
```

Here ⟨list of tables⟩ is a sequence of external symbols. In the list of tables, external symbols are separated by a comma. In this case, two different external symbols may have a common variable. To distinguish from which table the value of this variable should be chosen, ⟨list of arguments⟩ represents a sequence of expressions of the following type, separated by commas:

$$\langle\text{external symbol}\rangle.\langle\text{one of the variables assigned to this symbol}\rangle\,. \tag{1}$$

Here ⟨external symbol⟩ must be encountered in the sequence ⟨list of tables⟩. In this case, ⟨condition⟩ is a quantifier-free positive formula of signature Σ whose variables are given by (1), where ⟨external symbol⟩ is encountered in the sequence ⟨list of tables⟩. If a certain variable is assigned only to one of the external symbols included in ⟨list of tables⟩, one can just write this variable in the sequence ⟨list of arguments⟩ without specifying to what external symbol it is assigned.

Sometimes it is convenient to give new names to the variables of the resulting table. For example, to compare different fields of a row of the same table in the sequence ⟨list of tables⟩, it may be convenient to repeat the same external symbol several times. In these cases, one can apply the construction

```
select <list of arguments>
  from <list of tables>
  where <condition>
```

where ⟨list of tables⟩ is a sequence of expressions of the following type separated by commas:

$$\langle\text{external symbol}\rangle\ \langle\text{new name for this symbol}\rangle\,. \tag{2}$$

[1] In the definitions of SQL constructions, I often use the notation from the unpublished manuscript by S.M. Dudakov (see [23]). However, the situation considered here is, of course, different from that in [23].

Here ⟨new name for this symbol⟩ is a new previously nonencountered external symbol, and ⟨list of arguments⟩ is a sequence of expressions of the following form separated by commas:

$$\langle\text{new name of external symbol}\rangle.\langle\text{one of the variables assigned to this symbol}\rangle \text{ as } \langle\text{new name for this variable}\rangle\,. \quad (3)$$

Here ⟨new name for this variable⟩ is a variable that has not been used earlier. New names for variables should be chosen so that different variables get different new names; ⟨condition⟩ in this case is a quantifier-free positive formula of signature Σ whose variables have the form

$$\langle\text{new name of external symbol}\rangle.\langle\text{one of the variables assigned to this symbol}\rangle\,. \quad (4)$$

In this case, ⟨external symbol⟩ is encountered in the sequence ⟨list of tables⟩.

3.2. Grouping of data and aggregate functions. The grouping list is a list of fields separated by commas. The semantics of the operator `group by` is as follows: all the rows are combined into several groups so that each group is formed by the rows in which the values of all columns from the grouping list coincide. Then, a single row is formed from each group.

To choose a single value from each group for some field, or to generate a new value, one applies the following aggregate functions: `max`, for choosing the maximal value; `min`, for the minimal value; and `sum`, for finding the sum in the group of values of a coordinate that is not included in the grouping list. Another aggregate function, `count(*)`, is used for calculating the number of rows in a group.

If there are a finite number of rows in a group, then the definitions of these functions are obvious. Otherwise, only `min` is defined. Anyway, all aggregate functions are defined for finite tables.

In view of these arguments, the most general form of a query without subqueries is

```
select <list of arguments>
 from <list of tables>
 where <condition>
 group by <grouping list>
 having <condition1>
```
(5)

Here ⟨list of tables⟩ is defined, as before, as a sequence of expressions of the form (2) separated by commas, and ⟨grouping list⟩ is a sequence of expressions of the form (4) separated by commas. The expressions appearing in the grouping list are called elements of the grouping list; ⟨list of arguments⟩ is a sequence of expressions of the following forms separated by commas:

$$\langle\text{element of grouping list}\rangle \text{ as } \langle\text{new name for this variable}\rangle\,, \quad (6)$$

$$\langle\text{aggregate function}\rangle \text{ as } \langle\text{new name for this function}\rangle\,. \quad (7)$$

A variable appears in a query if and only if it has the form ⟨new name for this variable⟩ in expressions of the form (6) or (7) from the sequence ⟨list of arguments⟩. In this case, ⟨condition1⟩ is a quantifier-free positive formula of signature Σ whose variables have the form ⟨new name for this variable⟩ in expressions of the form (6) or (7) from the sequence ⟨list of arguments⟩. In this case, ⟨condition⟩ is a quantifier-free positive formula of signature Σ whose variables have the form (4), where ⟨external symbol⟩ is encountered in the sequence ⟨list of tables⟩.

3.3. Operations on queries.

3.3.1. *Union and intersection of queries.* The arguments in set-theoretic operations on queries are queries that contain the same number of variables. All the operations considered here have two

arguments each. It is assumed that each variable appears in the first argument if and only if it appears in the second.

The union of queries Query1 and Query2 is given by the query

```
(Query1)
 union
(Query2)
```

The second of the two expressions

```
((Query1)
 union
(Query2))
 union
(Query3)
```

```
(Query1)
 union
(Query2)
 union
(Query3)
```

is considered as a contraction of the first.

For the intersection of queries Query1 and Query2, the construction `intersect` is used:

```
(Query1)
 intersect
(Query2)
```

Just as in the case of union, when dealing with intersections of several queries, one may omit parentheses.

The resulting query obtained either by a union or by an intersection contains those and only those variables that appear in the first argument.

3.3.2. *Representations.* As before, the rank and the number of argument places are defined for each internal symbol. The ranks start from 1 and include a few consecutive natural numbers. There always exist internal symbols of rank 1. Internal symbols of rank $(i+1)$ exist only if there are internal symbols of rank i.

A representation of a query is created by an operator `create view` of the form

```
create view <name of representation> as
  <SQL-query>
```
(8)

or of the form

```
create view <name of representation> as
  not <SQL-query>
```
(9)

Command (9) requires that the name of a representation should be assigned a formula that is a negation of the formula assigned to the given SQL query.

The name of a representation is a new internal symbol. Its arity is the number of variables that appear in the query. A variable appears in a representation if and only if it appears in the query that creates this representation. We assume that there exists a representation for every query.

When defining a query in (5), one can include in the sequences ⟨list of tables⟩ not only external symbols of relations, but also the names of representations. Therefore, ⟨list of tables⟩ is now a sequence of expressions of the following forms, separated by commas:

$$\langle\text{external symbol}\rangle\ \langle\text{new name for this symbol}\rangle\,, \tag{10}$$

$$\langle\text{name of representation}\rangle\ \langle\text{new name for this representation}\rangle\,. \tag{11}$$

In the other definitions, one should write "⟨external symbol⟩ or ⟨name of representation⟩" instead of "⟨external symbol⟩."

This may give rise to recursive queries. However, the following conditions should be satisfied.

Define the rank of a representation as the rank of the name of this representation. A representation is used in another representation in cases (8) and (9) if the name of the former representation is encountered in the sequence ⟨list of tables⟩ for the latter representation. A representation is used in the union or in the intersection of representations if it is used in one of the representations that are united or intersected.

It is required that the rank of any representation be not less than the rank of each representation used in case (8) and be strictly greater than the rank of each representation used in case (9). The rank of the union or intersection of representations must be strictly greater than the rank of each argument.

3.4. Semantics. Recall that formulas assigned to elements of the sequence ⟨list of tables⟩ are called tables. If there are several tables, then their tensor product is a table in which the number of columns (the number of variables of the resulting formula) is equal to the sum of the numbers of columns of the multiplied tables, while each row is obtained by combining a row of the first table with a row of the second table, etc., and, finally, by combining with a row of the last table, and all such combinations appear in the tensor product. Here, it is assumed that the multiplied formulas do not have pairwise common variables, and the variables of the tensor product are all the variables of the multiplied tables and only those variables. This means that the tensor product of tables is defined by a conjunction of the formulas that define these tables.

A query is calculated step by step. The steps are numbered by pairs (i, j) of positive integers. The first step is numbered by the pair $(1, 1)$. At step (i, j), queries of rank i are calculated. The pairs (i, j) are ordered lexicographically.

Before the beginning of calculations (before the first step), all queries are assigned empty sets of tuples.

Suppose that before step (i, j) all queries are assigned quantifier-free positive formulas of signature Σ. At step (i, j), new values of the queries of rank i are calculated.

If a query of rank i is a union or an intersection of queries, then the resulting table contains those and only those rows that enter at least one table or, respectively, both tables simultaneously.

Suppose that a query of rank i is given by (5). First of all, we compose the tensor product of these tables. This tensor product has all the columns of the multiplied tables as its columns, while its rows are all possible rows obtained from a row of the first table by adding a row of the second table, etc., and, finally, by adding a row of the last table. From the table obtained, we take only those rows that satisfy the formula ⟨condition⟩. In the resulting table, we group (combine) rows with coinciding fields from the sequence ⟨grouping list⟩. In the table obtained, we leave only the columns from the list ⟨list of arguments⟩. In the table obtained, we leave only the rows that satisfy the formula ⟨condition1⟩. Then we combine the resulting table with the table assigned to the query under consideration and obtain a table assigned to this query after step (i, j).

If a representation of rank i has the form (8), then this representation is assigned the same table that is assigned to the query that creates this representation. If a representation of rank i has the form (9), then this representation is assigned the negation of the relation that is assigned to the query that creates this representation. Since the rank of this creating query is less than i, the formula assigned to this query has already been found before step (i, j) and, hence, the negation of this formula has already been found before this step. As already mentioned (Lemma 2.1), this negation is also equivalent to a quantifier-free positive formula of signature Σ.

If no table is changed at step (i, j), then the program goes to step $(i + 1, 1)$. Otherwise the program goes to step $(i, j + 1)$. If at step (i, j) it turns out that there are no internal symbols of

rank i, then the program terminates. The result of the program is the tables assigned to internal symbols.

Theorem 3.1. *After each calculation step (i, j), the formulas assigned to internal symbols are quantifier-free positive formulas of signature Σ.*

Proof. Suppose that, before step (i, j), all queries are assigned quantifier-free positive formulas of signature Σ. It remains to prove that the formulas obtained after step (i, j) are also quantifier-free positive formulas of signature Σ.

Indeed, the tensor product of tables is defined by a conjunction of formulas that define these tables; ⟨condition⟩ distinguishes from this conjunction the rows that are chosen. If we existentially quantify the formula obtained with respect to the columns that are not included in the ⟨list of arguments⟩, add columns corresponding to the included aggregate functions together with the definitions of these functions, and leave only the rows satisfying the formula ⟨condition1⟩, then we obtain the necessary formula. It remains to note that the aggregate functions are defined by quantifier-free positive formulas of signature Σ.

So, for a given quantifier-free positive formula $\phi(\overline{x}, \overline{y})$ of signature Σ and a given collection $\overline{a}$ of natural numbers, we find the least $\overline{b}$ satisfying $\phi(\overline{a}, \overline{b})$. The collections are ordered lexicographically.

First, we should write a rule that distinguishes, from among the collections $\overline{z}$ satisfying $P(\overline{x}, \overline{z})$, those for which this set contains a smaller set in the lexicographic order. The complement of this set distinguishes the minimal collection.

To determine the maximal collection, we should first write a rule that distinguishes, from among the collections $\overline{z}$ satisfying $P(\overline{x}, \overline{z})$, those for which this set contains a larger set in the lexicographic order. The complement of this set distinguishes the maximal collection. A set is finite if and only if a maximal collection exists.

Let $P(\overline{x}, \overline{y})$ be an external symbol for the formula $\phi(\overline{x}, \overline{y})$ (or an internal symbol of rank i for the representation to which this formula is assigned). Consider the rules

$$Q(\overline{x}, \overline{y}, \overline{u}) \leftarrow P(\overline{x}, \overline{z}) \;\; P(\overline{x}, \overline{y}) \;\; P(\overline{x}, \overline{u}) \;\; \overline{y} <^1 \overline{z} \;\; \overline{z} <^1 \overline{u};$$

$$\mathrm{NOT}\, Q(\overline{x}, \overline{y}, \overline{u}) \leftarrow P(\overline{x}, \overline{y}) \quad P(\overline{x}, \overline{u}) \;\; \overline{y} <^1 \overline{u} \;\; \neg Q(\overline{x}, \overline{y}, \overline{u}),$$

in which the internal symbol Q has rank i and the internal symbol $\mathrm{NOT}\, Q$ has rank $(i + 1)$. In these rules, $<^1$ denotes an internal symbol of rank 1 to which a relation of lexicographic comparison of collections is assigned. It is easy to define rules for calculating the values of this symbol. When the length of the collection $\overline{y}$ is 1, there is one rule, and the body of such a rule is given by $x < y$. It is clear that the validity of $\mathrm{NOT}\, Q(\overline{x}, \overline{y}, \overline{u})$ means that $\overline{u}$ is the collection following $\overline{y}$ in the lexicographic order in the set of collections $\overline{z}$ satisfying $P(\overline{x}, \overline{z})$.

Now, when the set of collections $\overline{z}$ satisfying $P(\overline{x}, \overline{z})$ is finite, we can calculate the number of such collections.

Preliminarily, we need the addition of natural numbers:

$$\mathrm{Sum}(x, 0, x) \leftarrow; \;\; \mathrm{Sum}(x, y, z) \leftarrow u =_1 y \;\; \mathrm{Sum}(x, u, v) \;\; v =_1 z.$$

To calculate the number of elements of the group, we introduce a function $\mathrm{NUMBER}(\overline{x}, \overline{y}, z)$. For the minimal $\overline{y}$, this number z is equal to 1. The next rule has the form

$$\mathrm{NUMBER}(\overline{x}, \overline{y}, z) \leftarrow \mathrm{NOT}\, Q(\overline{x}, \overline{u}, \overline{y}) \;\; \mathrm{NUMBER}(\overline{x}, \overline{u}, v) \;\; v =_1 z.$$

It is clear that $\mathrm{NUMBER}(\overline{x}, \overline{y}, z)$ for the maximal $\overline{y}$ yields z equal to the number of elements of the group. The sum of elements of a column for the group is calculated analogously.

This completes the proof of the fact that, after each step of calculation, all queries are assigned quantifier-free positive formulas of signature Σ. □

An SQL query and a stratified Datalog program are said to be equivalent if, for given values of external symbols, they either yield the same result or do not terminate simultaneously.

The above proves the following theorem.

Theorem 3.2. *For every* SQL *query, one can effectively construct an equivalent stratified Datalog program. In particular, if each external symbol is assigned a quantifier-free positive formula of signature* Σ_0 *and a given* SQL *query stops, then an answer is given by a collection of quantifier-free positive formulas of signature* Σ.

The converse assertion is quite obvious and is proved by induction on the maximal rank used in the program.

Theorem 3.3. *For every stratified Datalog program, one can effectively construct an equivalent* SQL *query.*

4. FIXED POINT LOGIC (FPL)

We will consider formulas of the signature $\Sigma_1 = \langle \Sigma, Q_1^{(n_1)}, \dots, Q_\ell^{(n_\ell)} \rangle$ that is obtained by enriching the signature Σ with additional n_i-ary symbols of relations Q_i for $i = 1, \dots, \ell$.

Let $\langle \Sigma_1, P^{(n)} \rangle$ be the signature obtained by enriching the signature Σ_1 with an additional n-ary symbol of relation P. We consider formulas of signature $\langle \Sigma_1, P^{(n)} \rangle$ in which the symbol P appears positively. Recall that the symbol P appears positively in a formula if this formula either has the form $P(z_1, \dots, z_n)$, where $z_1, \dots, z_n$ are variables, or is a disjunction of two formulas in each of which the symbol P appears positively, or is a conjunction of two formulas in each of which the symbol P appears positively, or this formula does not contain P.

We will consider algebraic systems of signature Σ_1 whose support is the set of natural numbers, the symbols Q_i for $i = 1, \dots, \ell$ are interpreted as quantifier-free positive formulas of signature Σ_0, and the conventions from the definition of a finite representation of a database are satisfied. In other words, we are given a scheme and a state of some database.

For this state and a formula $\phi(x_1, \dots, x_n)$ of signature $\langle \Sigma_1, P^{(n)} \rangle$ in which P appears positively, we define a formula $\mathrm{FP}(\phi)(x_1, \dots, x_n)$. We call the formula $\mathrm{FP}(\phi)(x_1, \dots, x_n)$ a fixed point of the formula $\phi(x_1, \dots, x_n)$ in the given state.

However, in fact we will consider a more general case. If an interpretation of a fixed point of the formula $\phi(x_1, \dots, x_n)$ is already defined and is a quantifier-free positive formula of signature Σ, then we extend the scheme of the database by adding an n-ary symbol of relation to the scheme Σ_1 and enrich the state assuming that the interpretation of this fixed point is the interpretation of this added symbol. The further definitions do not depend on at which step of extension of the database scheme we are.

An interpretation of the formula $\mathrm{FP}(\phi)(x_1, \dots, x_n)$ is constructed as follows. For a given state σ of the database, the interpretation of $\mathrm{FP}(\phi)(x_1, \dots, x_n)$ is $\bigcup_{i=1}^{\infty} P_i$, where P_0 is empty and

$$P_i = \big\{(a_1, \dots, a_n) \in \omega^n \mid (\sigma, P_{i-1}, a_1, \dots, a_n) \models \phi(x_1, \dots, x_n)\big\}.$$

Here the validity of $(\sigma, P_{i-1}, a_1, \dots, a_n) \models \phi(x_1, \dots, x_n)$ means the validity of $\phi(x_1, \dots, x_n)$ in (σ, P_{i-1}) for $x_1 = a_1, \dots, x_n = a_n$. As explained in [19], the system (σ, P_{i-1}) is an enrichment of σ in which the interpretation of the symbol P is given by the relation P_{i-1}.

According to the aforesaid, each P_i is defined by a quantifier-free positive formula of signature Σ. Since $\phi(x_1, \dots, x_n)$ contains P positively, P_{i+1} contains P_i for every i. Thus, if $P_{i+1} = P_i$ for some i, then the interpretation of the formula $\mathrm{FP}(\phi)(x_1, \dots, x_n)$ in the state considered is P_i for this i. If the sequence of these P_i strictly increases (which would be impossible on a finite support), then the value of the formula $\mathrm{FP}(\phi)(x_1, \dots, x_n)$ is not defined.

If the formula does not contain P, then its fixed point coincides with it in each state. Therefore, each formula can be considered as a fixed point of some formula. The formulas thus obtained are called *formulas of fixed point logic*.

The formulas can be ranked as follows. Formulas that do not contain fixed points have rank 0. The fixed points of formulas of rank 0 have rank 1. Formulas that contain fixed points of rank 1 but do not contain fixed points of higher rank have rank 1. The fixed points of formulas of rank 1 have rank 2. In general, formulas that contain fixed points of rank i but do not contain fixed points of higher rank have rank i. The fixed points of formulas of rank i have rank $(i+1)$.

If a formula contains fixed points, then the value of this formula may not be defined in some states. According to the above, if the value of a formula is defined on a certain state, then this value is a quantifier-free positive formula of signature Σ.

The following theorem is valid.

Theorem 4.1. *For every formula of fixed point logic, one can effectively construct a stratified Datalog program such that, for each state, the result of the program coincides with the value of the given formula. The program constructed does not terminate for the above-mentioned state if and only if the value of the given formula is not defined on this state.*

The proof of this theorem is routine and is carried out by induction on the rank of the formula considered.

The converse theorem is more informative. Its proof employs the fact, pointed out back in Moschovakis' book (see [16]), that induction on several parameters reduces in some cases to induction on a single parameter. Therefore, we present this proof. Although our construction is not something very original, in the present case it is incomparably simpler than Moschovakis's general arguments.

Theorem 4.2. *For every stratified Datalog program, one can effectively construct a set of formulas of fixed point logic such that, for every state, the result of the program coincides with the value of this set of formulas. Moreover, the program does not terminate for a given state if and only if the value of the constructed set of formulas is not defined on this state.*

Proof. Clearly, it suffices to consider the case of rules of rank i, assuming that, for internal symbols of lower rank, the formulas of fixed point logic that define the values of these symbols have already been constructed. Suppose given k internal symbols of rank i. Let m be the maximum possible number of arguments in these symbols. If some of these symbols has less than m arguments, we will assume that this symbol has precisely m arguments, repeating the last argument the necessary number of times. Introduce a new $(m+1)$-ary symbol Q of rank i.

Let Q_j be an m_j-ary symbol of rank i. We can assume that the head of every rule with name Q_j is $Q_j(x_1,\dots,x_{m_j})$. Replace every rule with name Q_j by a new rule with the head $Q(x_0,x_1,\dots,x_m)$ and a body that includes the atom $x_0=j$, all the atoms that appear in the body of the rule under consideration, and, in addition, all expressions of the following form that appear in the body of the rule under consideration:

- $Q(y_1,\dots,y_k)$, where Q is an external k-ary symbol and $y_1,\dots,y_k$ is a sequence of variables;
- $Q(y_1,\dots,y_k)$, where Q is an internal k-ary symbol whose rank is lower than i and $y_1,\dots,y_k$ is a sequence of variables;
- $\neg Q(y_1,\dots,y_k)$, where Q is an internal k-ary symbol whose rank is lower than i and $y_1,\dots,y_k$ is a sequence of variables.

As regards the expressions of the form $Q_\ell(z_1,\dots,z_{m_\ell})$ that appear in the body of the rule, where Q_ℓ is an m_ℓ-ary symbol of rank i, instead of these expressions we include the following sequences in the body of the new rule:

$$Q(z_0,z_1,\dots,z_{m_\ell},\dots,z_{m_\ell})\quad z_0=\ell.$$

Moreover, if $m > m_j$, we add the sequence

$$x_{m_{j+1}} = x_{m_j} \ \dots \ x_m = x_{m_j}$$

to the body of the new rule. After these transformations, we have exactly one external symbol of each rank.

Next, we replace the body of each rule by the conjunction of all elements of this body and then consider the disjunction of the conjunctions thus obtained for all rules with name Q, which we existentially quantify with respect to all the variables that do not enter the head of the rule. In this way we obtain a formula whose fixed point gives the value of the new symbol Q of rank i. To obtain the value of the old symbol Q_j, it suffices to add a conjunction with $x_0 = j$ and, if $m > m_j$, with $(x_{m_{j+1}} = x_{m_j} \,\&\, \dots \,\&\, x_m = x_{m_j})$. $\square$

ACKNOWLEDGMENTS

I am grateful to Lev Dmitrievich Beklemishev for his remarks that helped me to improve the original text of the manuscript, in particular, the original proof of Theorem 2.3.

This work was supported by the Russian Foundation for Basic Research, project no. 08-01-00241.

REFERENCES

1. S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases* (Addison-Wesley, Reading, MA, 1995).
2. S. Abiteboul, M. Y. Vardi, and V. Vianu, "Fixpoint Logics, Relational Machines, and Computational Complexity," in *Structure in Complexity Theory: Proc. 7th Annu. Conf., Boston, 1992* (IEEE Comput. Soc. Press, Los Alamitos, CA, 1992), pp. 156–168.
3. S. Abiteboul and V. Vianu, "Fixpoint Extensions of First-Order Logic and Datalog-like Languages," in *Logic in Computer Science: Proc. 4th Annu. Symp., Pacific Grove, CA, USA, 1989* (IEEE Comput. Soc. Press, Washington, DC, 1989), pp. 71–79.
4. F. Afrati, S. S. Cosmadakis, and M. Yannakakis, "On Datalog vs. Polynomial Time," in *Principles of Database Systems: Proc. 10th ACM SIGACT–SIGMOD–SOGART Symp., Denver, 1991* (Assoc. Comput. Mach., New York, 1991), pp. 13–25.
5. M. Ajtai and Y. Gurevich, "DATALOG vs. First-Order Logic," in *Foundations of Computer Science: Proc. 30th Annu. Symp.* (IEEE Comput. Soc. Press, Los Alamitos, CA, 1989), pp. 142–146.
6. O. V. Belegradek, A. P. Stolboushkin, and M. A. Taitslin, "On Problems of Databases over a Fixed Infinite Universe," in *Logic, Algebra, and Computer Science: Helena Rasiowa in Memoriam* (Pol. Acad. Sci., Inst. Math., Warszawa, 1999), Banach Center Publ. **46**, pp. 23–62.
7. A. Blass and Y. Gurevich, "Existential Fixed-Point Logic," in *Computation Theory and Logic*, Ed. by E. Börger (Springer, Berlin, 1987), Lect. Notes Comput. Sci. **270**, pp. 20–36.
8. A. Chandra and D. Harel, "Structure and Complexity of Relational Queries," J. Comput. Syst. Sci. **25**, 99–128 (1982).
9. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," Commun. ACM **13**, 377–387 (1970).
10. E. F. Codd, "Relational Completeness of Data Base Sublanguages," in *Data Base Systems*, Ed. by R. Rustin (Prentice-Hall, Englewood Cliffs, NJ, 1972), pp. 65–98.
11. N. Immerman, "Relational Queries Computable in Polynomial Time," Inf. Control **68**, 86–104 (1986).
12. N. Immerman, "Languages That Capture Complexity Classes," SIAM J. Comput. **16**, 760–778 (1987).
13. P. C. Kanellakis and D. Q. Goldin, "Constraint Programming and Database Query Languages," in *Theoretical Aspects of Computer Software: Proc. 2nd Int. Symp. TACS'94* (Springer, Berlin, 1994), Lect. Notes Comput. Sci. **789**, pp. 96–120.
14. P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz, "Constraint Query Languages," J. Comput. Syst. Sci. **51** (1), 26–52 (1995).
15. L. Libkin, *Elements of Finite Model Theory* (Springer, Berlin, 2004).
16. Y. N. Moschovakis, *Elementary Induction on Abstract Structures* (North-Holland, Amsterdam, 1974).
17. A. P. Stolboushkin and M. A. Taitslin, "Safe Stratified Datalog with Integer Order Does Not Have Syntax," ACM Trans. Database Syst. **23** (1), 100–109 (1998).

18. C. J. Date, *An Introduction to Database Systems*, 8th ed. (Pearson Education, Boston, MA, 2003; Vil'yams, Moscow, 2005).
19. S. M. Dudakov and M. A. Taitslin, "Collapse Results for Query Languages in Database Theory," Usp. Mat. Nauk **61** (2), 3–66 (2006) [Russ. Math. Surv. **61**, 195–253 (2006)].
20. K. E. Kline, *SQL: In a Nutshell. A Desktop Quick Reference*, 2nd ed. (O'Reilly, Sebastopol, CA, 2004; Kudits-Obraz, Moscow, 2006).
21. P. Wilton and J. Colby, *Beginning SQL* (Wiley, Indianapolis, IN, 2005; Dialektika, Moscow, 2006).
22. B. Forta, *Sams Teach Yourself SQL in 10 Minutes*, 3rd ed. (Sams Publ., Indianapolis, IN, 2004; Vil'yams, Moscow, 2005).
23. S. M. Dudakov, "SQL," http://homepages.tversu.ru/~p000101/dudakov.pdf

Translated by I. Nikitin