Computing Aggregations in Database Query Languages[⋆]

Luca Cabibbo and Riccardo Torlone

Università Roma Tre, Dipartimento di Informatica e Automazione, Italy
{cabibbo,torlone}@dia.uniroma3.it

Abstract. In this paper we present a framework for studying aggregations in the context of database query languages. Starting from a broad definition of aggregate function, we address our investigation from two different perspectives. We first propose a *declarative* notion of uniform aggregate function that refers to a family of scalar functions uniformly constructed over a vocabulary of basic operators by a bounded Turing Machine. This notion yields an effective tool to study the embedding of *classes* of built-in aggregate functions in query languages. All the aggregate functions most used in practice are included in this classification. We then present an *operational* notion of aggregate function, by considering a high-order folding constructor, based on structural recursion and partially implemented in commercial database systems, devoted to compute numeric aggregations over complex values. We show that numeric folding over a given vocabulary is sometimes not able to compute, by itself, the whole class of uniform aggregate function over the same vocabulary. It turns out however that this limitation can be partially remedied by the restructuring capabilities of query languages.

1 Introduction

Computing aggregations has been always considered an important feature of practical database query languages. This ability is indeed fundamental in specific application domains whose relevance has recently increased; among them, on-line analytical processing (OLAP), decision support, statistical evaluation, and management of geographical data. In spite of this fact, a systematic study of aggregations in the context of query languages has evolved quite slowly. Apart from the work by Klug [14], who formalized extensions of algebra and calculus with aggregates, in the last decade there have been few papers dealing with this subject [7, 15–17], even if specific aggregate functions of theoretical relevance, like counters, have received special attention [2, 8, 11].

The general approach to the problem is to study basic properties of formal languages equipped with a specific class of built-in aggregate functions (typically the ones provided by SQL, that is, MIN, MAX, SUM, COUNT and AVG, plus further functions of theoretical interest, like EVEN). Conversely, we would like to attack the problem from a more general perspective, possibly independent of the specific aggregate functions chosen. To do that, we first need to answer a fundamental question: *what is an aggregate function?* It is folk knowledge that a database aggregate function takes a collection of objects (with or without duplicates) as argument, and returns a new value that "summarizes" a numeric property of the collection. This broad definition is clearly too vague to provide clues for answering general questions about query languages with aggregation capabilities.

[⋆] An extended version of this paper has been published under the title "A Framework for the Investigation of Aggregate Functions in Database Queries" in *International Conference on Database Theory, 1999.* This work was partially supported by *CNR* and by *MURST*.

Our first goal is then trying to refine this definition, in order to provide a solid basis to the whole picture. Borrowing some ideas from the circuit model [13, 19], we start by noting that an aggregate function g over a collection s can be effectively described by a family of *scalar* functions (that is, traditional k-ary functions) $G = \{g_0, g_1, g_2, \ldots\}$ such that, for each $k \geq 0$: (i) g_k computes g when s contains exactly k elements; (ii) the result of g_k is invariant under any permutation of its arguments (as g operates on collections rather than sequences); and (iii) g_k is constructed by using a fixed vocabulary of basic operations and constants. To guarantee tractability of this representation in terms of families of scalar functions, we need to set some constraint on how the various g_k's are built. We make this constraint precise by introducing a notion of *uniform construction*, stating that a circuit description of each function g_k in G needs to be "easily" generated from k (specifically, from a Turing machine having bounded complexity). In this way, the family G forms a good representative of the aggregate function g, since the computational power is captured by the g_k's themselves rather than by their construction. This approach leads to the definition, in a very natural way, of different and appealing *abstract classes* of aggregate functions: a class is composed of all the aggregate functions that can be represented by a uniform family of functions over a given vocabulary. With the assumption that each scalar function in the vocabulary has constant computational cost, uniform construction indeed guarantees tractability of the aggregate functions so defined.

Our second goal is to address the impact of incorporating aggregate functions in a database query language. To this end, we consider a two-sorted algebra for complex values [1], called $\mathcal{CVA}$, over an uninterpreted domain and an interpreted one. In this context, we first study extensions of $\mathcal{CVA}$ with built-in aggregate functions, that is, with operators whose semantics is defined outside the database. We then present a simple constructor, called *folding*, that allows the user to define and apply an aggregation *within* $\mathcal{CVA}$. The folding operator is based on structural recursion [5] and is defined in terms of a pre-function p, an increment function i over the variables acc and $curr$, and a post-function q. The application of a folding expression over a collection s of complex values returns a new value by iterating over the elements of s in a natural way: starting from $acc = p(s)$, for each element $curr$ of s (chosen in some arbitrary order) the function i is applied to acc and $curr$; the result is obtained by applying q to the value of acc at the end of the iteration. A constraint on i guarantees that the result of folding over a collection s is indeed independent of the order in which the elements of s are selected during the evaluation. We then restrict our attention to a simplified version of folding devoted to compute *numeric* aggregations: it operates only on multisets of interpreted atomic values and returns a numeric value, without involving complex values in its evaluation. The folding operator is thus similar in spirit to User-Defined Aggregates [12] in SQL3, but it is tailored to numeric aggregations in a complex-values data model.

We finally relate the abstract notion of uniform aggregate function with this procedural way of computing aggregations. We first show that, even if numeric folding is less expressive than general folding, they have the same expressive power when embedded in $\mathcal{CVA}$; that is, the restructuring capabilities of the algebra overcome the gap in expressiveness. We then show that numeric folding over a given vocabulary of scalar functions computes only uniform aggregate functions over the same vocabulary, but not all of them. We demonstrate that this limitation can be partially overcome by the restructuring capabilities of a query language. It turns out however that $\mathcal{CVA}$, extended with the folding operator and a vocabulary of scalar functions, is still not able to capture the whole class of uniform aggregate functions over the given vocabulary.

The rest of the paper is devoted to a formalization of the issues discussed in this section and is organized as follows. In Section 2, we introduce the basic notions and present a number of examples of uniform aggregate functions. In Section 3, we introduce the $\mathcal{CVA}$ query language and investigate extensions of $\mathcal{CVA}$ with built-in aggregate functions. The folding operator is introduced in Section 4, where an important restriction of it is also characterized. In Section 5, we relate the expressive power of the various extensions of $\mathcal{CVA}$ with aggregations. Finally, in Section 6, we state a number of interesting open problems.

2 Aggregate Functions

2.1 Basic Definitions

We first introduce a very general definition of aggregate function. Let $\{\!\!\{\mathcal{N}\}\!\!\}$ denote the class of *finite multisets* of values from a countably infinite domain $\mathcal{N}$. (Multisets generalize sets, in that they allow an element to occur multiple times.)

Definition 1. *An* aggregate function *over $\mathcal{N}$ is a total function from $\{\!\!\{\mathcal{N}\}\!\!\}$ to $\mathcal{N}$, mapping each multiset of values to a value.*

Examples of aggregate functions over numeric domains are sum ($\sum$), product ($\prod$), counting, average, maximum, and minimum. Maximum and minimum are also examples of aggregate functions over collections of (non-numeric) totally ordered domains like strings. Note that we require aggregate functions to be *total*; in some cases, this requires some attention. For instance, the sum of a multiset of numbers is always well-defined; conversely, in defining the maximum function MAX, an arbitrary choice for its result over the empty multiset $\{\!\!\{\}\!\!\}$ is required.

In the rest of the paper, we will only consider aggregate functions over the domain $\mathbb{Q}$ of the rational numbers.

An important aspect of our approach is that we clearly separate the restructuring capabilities of a query language from its ability to compute aggregations. Specifically, we make a distinction between *aggregate functions* (that is, aggregations over numbers) and *aggregate queries* (that is, queries involving aggregate functions). Under this interpretation, the counting of a set of numbers is an *aggregate function*, whereas the counting of an arbitrary set (say, of a set of strings) is an *aggregate query*, which can be computed by first mapping each element to some numeric constant (say, 1), and then applying the aggregate function that counts the elements of the resulting numeric multiset. Similarly, testing whether two sets are equinumerous can be viewed as an aggregate query accomplished by computing the cardinalities of the sets with an aggregate function, and then checking for their equality. Finally, we note that the maximum function with SQL semantics is an aggregate query that returns a singleton set over non-empty sets and an empty set over empty sets; again, this can be implemented by using a maximum aggregate function together with some restructuring operations.

2.2 Uniform Aggregate Functions

In order to provide a concrete basis for the investigation of aggregations in the context of database queries, we now refine the definition of aggregate function. We first introduce a number of preliminary notions, to develop the following ideas, inspired from the circuit model [13, 19]: (i) an aggregate function can be represented by a family of scalar functions; (ii) each scalar function can be described by an arithmetic circuit over a collection of base functions[1]; (iii) uniformity in the construction of a family of circuits guarantees tractability of the represented aggregate function.

A *scalar function* (over the domain $\mathbb{Q}$ of the rational numbers) is a total function from $\mathbb{Q}^k$ to $\mathbb{Q}$, with $k \geq 0$. (Examples of scalar functions are the nullary functions 0 and 1 and the binary arithmetic functions $+$, $*$, $-$, and $/$.) An *enumeration* of a multiset $s = \{\!\{v_1, \ldots, v_n\}\!\}$ is a tuple $\vec{s} = [v_1, \ldots, v_n]$ containing the same elements as s with the same multiplicities, in any of the possible orderings. A *family of functions* is a set G of scalar functions such that, for each $k \geq 0$, there is one function $g_k : \mathbb{Q}^k \to \mathbb{Q}$ in G, called the *k-th component of G*.

We say that a family G of functions *represents* an aggregate function $h : \{\!\{\mathbb{Q}\}\!\} \to \mathbb{Q}$ if, for each $k \geq 0$, each multiset s of cardinality k, and each enumeration $\vec{s}$ of s, it is the case that $g_k(\vec{s}) = h(s)$, where g_k is the k-th component of G. Note that the above definition implies that only *symmetric* functions can be used in the representation of aggregate functions. We recall that a k-ary function f is symmetric if it remains invariant under any permutation of its arguments, that is, $f(x_1, \ldots, x_k) = f(x_{\sigma(1)}, \ldots, x_{\sigma(k)})$ for every permutation σ on $\{1, \ldots, k\}$.

As an example, consider the aggregate function SUM such that SUM(s) is the sum of the elements in the multiset s. The function SUM is represented by the following family of functions:

$$\{\text{SUM}_k \mid \text{SUM}_k(v_1, \ldots, v_k) = v_1 + \ldots + v_k\}.$$

Let a *vocabulary* be a set of scalar functions over $\mathbb{Q}$. A *circuit* over a vocabulary is a labeled directed acyclic graph whose nodes are either *function nodes* or are *input nodes*. Each function node is labeled by a function in the vocabulary; a node labeled by a m-ary function has precisely m ingoing arcs (the arcs are ordered). An *input node* is a node with no ingoing arcs labeled by x_i, with $i > 0$. There is one distinguished node with no outgoing arcs, called the *output* of the circuit. A *k-ary circuit* is a circuit with at most k input nodes having different labels from $x_1, \ldots, x_k$. The *semantics* of a k-ary circuit is the k-ary scalar function computed by the circuit in the natural way. A circuit *describes* a scalar function f if its semantics coincides with f. An *encoding* of a circuit is a listing of its nodes, with the respective labels. The *size* of the circuit is the number of its nodes. A *description* of a function f, together with its size, is a circuit that describes f.

In what follows, we shall often consider the vocabulary $\Omega_{ari} = \{0, 1, +, -, *, /\}$ that includes the basic arithmetic functions. Examples of circuits over Ω_{ari} are reported in Figures 1-3. For instance, the rightmost circuit in Figure 2 is a binary circuit that describes the scalar function $\text{SUM}_2(x_1, x_2) = x_1 + x_2$, which as been implemented as $x_2 + (x_1 + 0)$. As further examples, the rightmost circuits in Figures 1 and 3 describe the functions $\text{COUNT}_2(x_1, x_2) = 2$ and $\text{AVG}(x_1, x2) = (x_1 + x_2)/2$, respectively.

To guarantee the tractability of a family of functions, we introduce a notion of uniformity. A family G of functions is *uniform* if a description of the n-th component of G can be generated by a deterministic Turing machine using $O(\log n)$ workspace on input 1^n.

Definition 2. *A* uniform aggregate function *over a vocabulary Ω is an aggregate function that can be represented by a uniform family of functions over Ω.*

[1] Actually, there are other ways to describe scalar functions, such as *straight line programs* and *formulae*, which are essentially equivalent to arithmetic circuits.

As a particular but important case, we call *uniform counting functions* the uniform aggregate functions that can be represented by families of constant functions. We note that, since a constant function does not depend on its arguments, a uniform counting function can be described by a family of circuits with input nodes.

It is worth noting that we use the term "uniform" to mean *logspace uniform*, as it is customary in the context of the circuit model, where several different notions of uniformity are essentially equivalent to *logspace* uniformity. Note also that, as a consequence of the requirement of *logspace* computability in the definition, the size of the description of the n-th component of any uniform family is polynomial in n.

As first examples, Figures 1-3 show the first three components of the families that describe the uniform aggregate functions COUNT, SUM, and AVG (average), respectively, over the vocabulary Ω_{ari}. Among them, function COUNT is a counting function.

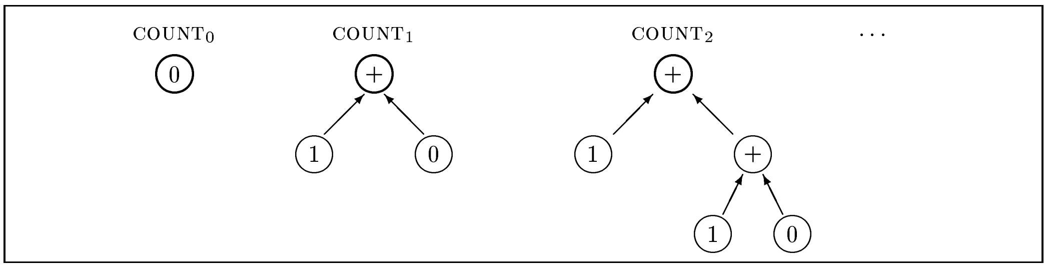

Fig. 1. A circuit representation of the aggregate function COUNT

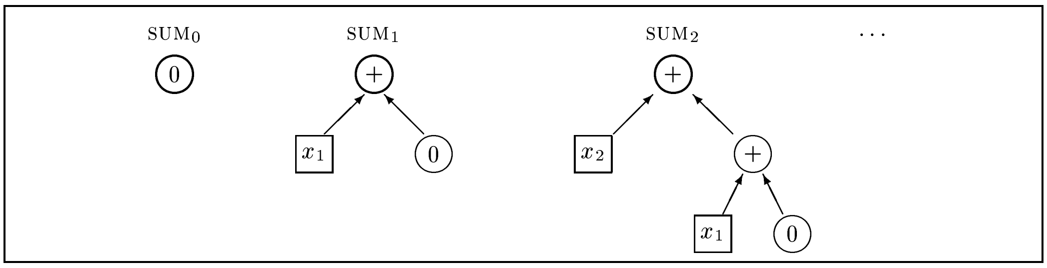

Fig. 2. A circuit representation of the aggregate function SUM

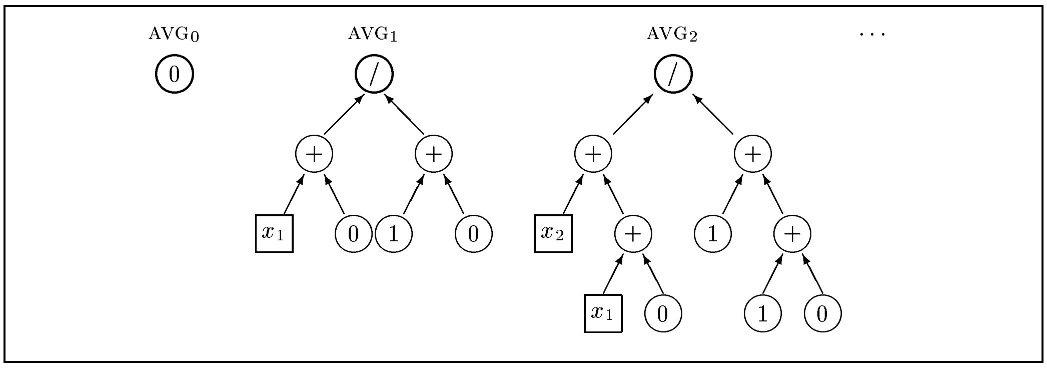

Fig. 3. A circuit representation of the aggregate function AVG

We note that, for each $k > 0$, the left and right subtrees of the output of AVG$_k$ correspond to the k-th component of the aggregate functions SUM and COUNT, respectively. Actually, these three aggregate functions can be defined inductively as follows:

$$\begin{array}{rlrll} \textsc{sum}_0 = 0 & & \textsc{sum}_k = \textsc{sum}_{k-1} + x_k & & \text{for } k > 0 \\ \textsc{count}_0 = 0 & & \textsc{count}_k = \textsc{count}_{k-1} + 1 & & \text{for } k > 0 \\ \textsc{avg}_0 = 0 & & \textsc{avg}_k = \textsc{sum}_k / \textsc{count}_k & & \text{for } k > 0 \end{array}$$

Intuitively, the fact that the k-th component of a family can be derived from the previous component in a simple way guarantees uniformity of the family [13].

The vocabulary Ω_{ari} can be used to define several other aggregate functions. For example, the function EVEN can be defined as $\textsc{even}_0 = 1$ and $\textsc{even}_k = 1 - \textsc{even}_{k-1}$, for each $k > 0$, in which 1 is interpreted as *true* and 0 as *false*. A different family for the same function is such that $\textsc{even}'_{2k} = 1$ and $\textsc{even}'_{2k+1} = 0$, for each $k \geq 0$. The aggregate function EXP, such that $\textsc{exp}_k = 2^k$, can be defined as $\textsc{exp}_0 = 1$ and $\textsc{exp}_k = \mathit{double}(\textsc{exp}_{k-1})$ for $k > 0$, where $\mathit{double}(x) = x + x$. Note that the obvious description $1 + 1 + \ldots + 1$ for $\textsc{exp}_k$ (with 2^k number of 1's and $2^k - 1$ number of $+$'s) cannot be generated by a *logspace* computation, since it involves a number of operators which is exponential with respect to k. Further uniform aggregate functions over Ω_{ari} are the following: any fixed natural or rational constant c; the function FATT, such that $\textsc{fatt}_k = k!$; the function PROD, such that $\textsc{prod}_k = x_1 * \ldots * x_k = \prod_{i=1}^{k} x_i$; the function FIB that generates Fibonacci's numbers.

Functions like COUNT, EVEN, and EXP do not depend on the actual values of the elements in the argument (multi)set, but rather on the number of elements it contains. According to our definition, these are indeed uniform counting functions.

Although many interesting functions can be defined in terms of Ω_{ari}, this vocabulary does not allow the definition of the minimum and maximum functions MIN and MAX. To this end, we need to use further scalar functions to take care of order. For instance, using the functions in Ω_{ari} plus an *ordering* function $\geq : \mathbb{Q} \times \mathbb{Q} \to \{0, 1\}$ it is easy to define also the scalar functions $=$, $\neq$, and $>$. Then, it can be shown that MIN and MAX are uniform aggregate functions over $\Omega_{ari} \cup \{\geq\}$.

2.3 A Hierarchy of Aggregate Functions

Let $\mathcal{A}(\Omega)$ be the class of uniform aggregate functions over a vocabulary Ω. According to our definition, an element of $\mathcal{A}(\Omega)$ can be represented by a uniform family of functions whose description of the n-th component can be generated by a Turing machine using $O(\log n)$ workspace. Actually, a Turing machine using $O(\log n)$ workspace is essentially equivalent to a program that uses a fixed number of counters, each ranging over $\{1, \ldots, n\}$; it is then apparent that the expressiveness of such a machine increases with the number of available counters. To capture this fact, we now introduce a hierarchy of classes of uniform aggregate functions, as follows.

Definition 3. *$\mathcal{A}^k(\Omega)$ is the class of uniform aggregate functions such that the description of the n-th component has size $O(n^k)$.*

For instance, many of the aggregate functions introduced in Section 2.2 (including COUNT, SUM and AVG) belong to $\mathcal{A}^1(\Omega_{ari})$.

From the above definition, it easily follows that $\mathcal{A}(\Omega) = \bigcup_{k \geq 0} \mathcal{A}^k(\Omega)$ and that $\mathcal{A}^k(\Omega) \subseteq \mathcal{A}^{k+1}(\Omega)$ for each $k \geq 0$. Actually, it turns out that there are vocabularies for which the above hierarchy is proper.

Lemma 1. *There are a vocabulary Ω and a natural k such that $\mathcal{A}^k(\Omega) \subset \mathcal{A}^{k+1}(\Omega)$.*

Sketch of proof: Let $\Omega_{abs} = \{0_\oplus, \oplus, \otimes\}$ be an "abstract" vocabulary, where $\oplus$ is a commutative binary function, the constant $0_\oplus$ is an identity element for $\oplus$, and $\otimes$ is an arbitrary binary function. Consider the uniform aggregate function α over Ω_{abs} such that $\alpha_n = \bigoplus_{i=1,\ldots,n}^{j=1,\ldots,n} x_i \otimes x_j$, where $\bigoplus$ is defined in terms of $\oplus$ and $0_\oplus$ as $\bigoplus(v_1, \ldots, v_k) = 0_\oplus \oplus v_1 \oplus \ldots \oplus v_k$. Since the n-th component α_n contains $O(n^2)$ operators, the function α is in $\mathcal{A}^2(\Omega_{abs})$ by definition. Moreover, α does not belong to $\mathcal{A}^1(\Omega_{abs})$, since it is not possible to represent its components using a linear number of operators. Thus, $\mathcal{A}^1(\Omega_{abs}) \subset \mathcal{A}^2(\Omega_{abs})$. $\Box$

Remark 1. Let us now briefly discuss the above result. Consider the following "concrete" interpretation $\widetilde{\Omega}_{abs}$ for the abstract vocabulary Ω_{abs}: $0_\oplus$ is 0 (the rational number zero), $\oplus$ is $+$ (addition of rational numbers), and $\otimes$ is $*$ (multiplication of rational numbers). According to this interpretation $\widetilde{\Omega}_{abs}$, because multiplication is distributive over addition, the n-th component $\alpha_n = \sum_{i,j} x_i * x_j$ of α can be rewritten as $(\sum_i x_i) * (\sum_j x_j)$, which has $O(n)$ operators, and therefore α belongs to $\mathcal{A}^1(\widetilde{\Omega}_{abs})$. Conversely, consider the function β such that $\beta_n = \bigoplus_{i,j}^{i \neq j} x_i \otimes x_j$. It is easy to see that β belongs to $\mathcal{A}^2(\Omega_{abs}) - \mathcal{A}^1(\Omega_{abs})$. Furthermore, even with respect to the interpretation $\widetilde{\Omega}_{abs}$ for the vocabulary Ω_{abs}, the function β belongs to $\mathcal{A}^2(\widetilde{\Omega}_{abs}) - \mathcal{A}^1(\widetilde{\Omega}_{abs})$. It would belong to $\mathcal{A}^1(\widetilde{\Omega}_{abs})$ if $\widetilde{\Omega}_{abs}$ contained the difference operator $-$; in this case, β_n could be rewritten as $(\sum_i x_i) * (\sum_j x_j) - (\sum_k x_k * x_k)$. Thus, in general, properties of the aggregate functions in the class $\mathcal{A}(\Omega)$ depend on properties of the scalar functions in the vocabulary Ω. $\Box$

3 A Query Language with Interpreted Functions

In this section we investigate the embedding of a class of aggregate functions within a database query language. We first introduce an algebra for complex values [1], and then extend it with interpreted scalar and aggregate functions.

3.1 The Data Model

We consider a two-sorted data model for complex values, over a countably infinite, uninterpreted domain $\mathcal{D}$ and the interpreted domain $\mathbb{Q}$ of the rational numbers. We fix two further countably infinite disjoint sets: a set $\mathcal{A}$ of *attribute names* and a set $\mathcal{R}$ of *complex-values names*. The *types* of the model are recursively defined as follows: (i) $\mathcal{D}$ and $\mathbb{Q}$ are *atomic* types; (ii) if $\tau_1, \ldots, \tau_k$ are types and $A_1, \ldots, A_k$ are distinct attribute names from $\mathcal{A}$, then $[A_1 : \tau_1, \ldots, A_k : \tau_k]$ is a *tuple* type ($k \geq 0$); and (iii) if τ is a type, then $\{\tau\}$ is a *set* type. The domain of complex values associated with a type is defined in the natural way.

A *database scheme* is a tuple of the form $(s_1 : \tau_1, \ldots, s_n : \tau_n)$, where $s_1, \ldots, s_n$ are distinct complex-values names from $\mathcal{R}$ and each τ_i is a type. A *database instance* is a function mapping each complex-values name s_i to a value of the corresponding type τ_i, for $1 \leq i \leq n$. Note that the s_i's are not required to be *sets* of (complex) values.

3.2 The Complex-Values Algebra

Our reference language, denoted by $\mathcal{CVA}$, is a variant of the complex-values algebra of Abiteboul and Beeri [1] without *powerset*. The language is based on *operators*, similar in spirit to relational algebra operators, and *function constructors*, which are high-order constructors used to apply functions to complex values. We now briefly recall the main features of the language, referring the reader to [1] for further details.

The expressions of the language describe functions[2] from a database scheme to a certain type, which are built by combining operators and function constructors starting from a basic set of functions. More specifically, the *base functions* include constants and complex-values names from $\mathcal{R}$ (viewed as nullary functions), attribute names from $\mathcal{A}$, the identity function id, the set constructor $\{\}$, the binary predicates $=$, $\in$, and $\subseteq$, the boolean connectives $\wedge$, $\vee$, and $\neg$. The operators include the set operations $\cup$, $\cap$, and $-$, the cross product (which is a variant of the k-ary Cartesian product), and the set-collapse (which transforms a set of sets into the union of the sets it contains).

The function constructors allow the definition of further functions, as follows:

– the *binary composition* constructor $f \circ g$, where f and g are functions, defines a new function whose meaning is "apply g, then f";
– the *labeled tuple* constructor $[A_1 = f_1, \ldots, A_n = f_n]$, where $f_1, \ldots, f_n$ are unary functions and $A_1, \ldots, A_n$ are distinct attribute names, defines a new function over any type as follows:

$$[A_1 = f_1, \ldots, A_n = f_n](s) = [A_1 : f_1(s), \ldots, A_n : f_n(s)];$$

– the *replace* constructor $\mathit{replace}\langle f \rangle$, where f is a unary function, defines a new function over a set type as: $\mathit{replace}\langle f \rangle(s) = \{f(w) \mid w \in s\}$;
– the *selection* constructor $\mathit{select}\langle c \rangle$, where c is a unary boolean function, defines a new function over a set type as: $\mathit{select}\langle c \rangle(s) = \{w \mid w \in s \text{ and } c(w) \text{ is true}\}$.

To increase readability, we will often use an intuitive, simplified notation. For example, if A and B are attributes and f is a function, we write $A.B$ instead of $B \circ A$, $A(f)$ instead of $A \circ f$, and $A \in B$ instead of $\in\circ[A, B]$. For the labeled tuple constructor, we write $[A]$ instead of $[A = A]$, and $[A.B]$ instead of $[B = A.B]$ (that is, an attribute takes its name from the name of the last navigated attribute).

As an example, let s_1 and s_2 be complex-values names, having types $\{[A : \mathcal{D}, B : \mathcal{D}]\}$ and $\{[E : \{\mathcal{D}\}, F : \mathcal{D}]\}$, respectively. Then the following expression computes the projection on A and F of the join of s_1 with s_2 based on the condition $B \in E$:

$$\mathit{replace}\langle [X.A, Y.F] \rangle(\mathit{select}\langle X.B \in Y.E \rangle(\mathit{cross}_{[X,Y]}(s_1, s_2))).$$

3.3 Adding Interpreted Functions

Interpreted functions (that is, functions whose semantics is defined outside the database) can be included in $\mathcal{CVA}$ in a very natural way [1].

Let us first consider a set Ω of interpreted scalar functions over $\mathbb{Q}$. These functions can be embedded into $\mathcal{CVA}$ by simply extending the set of available base functions with those in Ω, and allowing their application through the function constructors. For instance, if $+$ is in Ω, we can extend the tuples of a complex value r of type $\{[A : \mathbb{Q}, B : \mathbb{Q}]\}$ with a further attribute C holding the sum of the values in A and B by means of the expression $\mathit{replace}\langle [A, B, C = A + B] \rangle(r)$.

[2] We mainly refer to a general notion of *function* rather than *query* because, in order to include aggregations, it is convenient to relax the assumption $\mathcal{CVA}$ expressions evaluate to sets, to permit complex-values of any type as results.

Let us now consider a set Γ of aggregate functions over $\mathbb{Q}$. In this case we cannot simply extend the base functions with those in Γ, since we could obtain incorrect results. For example, if we need to sum the A components of the tuples in the complex value r above, we cannot use the expression $\textsc{sum}(\mathit{replace}\langle A\rangle(r))$, since this would eliminate duplicates before the aggregation. Therefore, we introduce an *aggregate application* constructor $g\{\!\{f\}\!\}$, where g is an aggregate function in Γ and f is any $\mathcal{CVA}$ function. For a set s, $g\{\!\{f\}\!\}(s)$ specifies that g has to be applied to the multiset $\{\!\{f(w) \mid w \in s\}\!\}$ rather than to the set $\{f(w) \mid w \in s\}$. Thus, the above function can be computed by means of the expression $\textsc{sum}\{\!\{A\}\!\}(r)$.

In the following, we will denote by $\mathcal{CVA} + \Omega + \Gamma$ the complex-values algebra extended in this way with the scalar functions in Ω and the aggregate functions in Γ.

3.4 Expressive Power and Complexity

It is well-known that the complex-values algebra $\mathcal{CVA}$ is equivalent to a lot of other algebraic or calculus-based languages (without *powerset*) over complex values and nested collections proposed in the literature [5, 6, 20, 21]. It turns out that $\mathcal{CVA}$ expresses only functions (over uninterpreted databases) that have PTIME data complexity.

When considering the complexity of functions over numeric interpreted domains, a cost model has to be specified, since it can be defined in several different ways. For instance, it is possible to consider a number as a bit-sequence and define the complexity of a function over a tuple of numbers with respect to the length of the representations of the numbers. However, we prefer to treat numbers as atomic entities, and assume that basic scalar functions are computed in a single step, independently of the magnitude or complexity of the involved numbers [4, 10]. Specifically, we assume that the computational cost of any scalar function that we consider is unitary; this is indeed a reasonable choice when arithmetic operations and comparisons over the natural or rational numbers are considered. Under this assumption, the data complexity of $\mathcal{CVA} + \Omega$ is in PTIME. Furthermore, if we consider a class Γ of aggregate functions having polynomial time complexity in the size of their input then, by a result in [5], $\mathcal{CVA} + \Omega + \Gamma$ remains in PTIME.

By noting that, for a collection Ω of scalar functions, the complexity of evaluating uniform aggregate functions in $\mathcal{A}(\Omega)$ is polynomial in the size of their input, the following result easily follows.

Theorem 1. *Let Ω be a collection of scalar functions. Then $\mathcal{CVA} + \Omega + \mathcal{A}(\Omega)$ has* PTIME *data complexity.*

4 An Operator for Defining Aggregate Functions

In this section we investigate the introduction, in the query language $\mathcal{CVA}$, of a high-order constructor, called *folding*, that allows us to *define* and *apply* an aggregation. This operator is essentially based on *structural recursion* [5, 18] and similar in spirit to User-Defined Aggregates [12] in SQL3.

4.1 Folding Expressions

A *folding signature* is a triple (τ_i, τ_a, τ_o) of types, with the restriction that the type τ_a does not involve the set type constructor. As we will clarify later, this type restriction ensures tractability of folding.

Definition 4. *A* folding constructor *of signature* (τ_i, τ_a, τ_o) *is an expression of the form* $\mathit{fold}\langle p; i; q\rangle$*, where:*

- p *is a* $\mathcal{CVA}$ *function of type* $\{\!\{\tau_i\}\!\} \to \tau_a$*, called* pre-function*;*
- i *is a left-commutative*[3] $\mathcal{CVA}$ *function* $\tau_i \times \tau_a \to \tau_a$ *over the symbols curr and acc, called* increment function*; and*
- q *is a* $\mathcal{CVA}$ *function of type* $\tau_a \to \tau_o$*, called* post-function.

A folding $\mathit{fold}\langle p; i; q\rangle$ of signature (τ_i, τ_a, τ_o) defines a function of type $\{\!\{\tau_i\}\!\} \to \tau_o$; the result of applying this function over a collection s is computed iteratively as follows:

```
$acc := p(s)$;
**for each** $curr$ **in** $s$ **do** $acc := i(curr, acc)$;
**return** $q(acc)$;
```

Initially, the result of applying the pre-function p to s is assigned to the "accumulator" variable *acc*. Then, for each element *curr* of s (chosen in some arbitrary order), the result of applying i to *curr* and *acc* is (re)assigned to *acc* (*curr* stands for current element). Finally, the result of the whole expression is given by the application of the post-function q to the value of *acc* at the end of the iteration. It is important to note that, since the function i is left-commutative, the semantics of folding is well-defined, that is, the result is independent of the order in which the elements of s have been selected.

For example, let r be a complex value of type $\{[A : \mathcal{D}, B : \mathbb{Q}]\}$. The count of the tuples in r can be computed by the expression: $\mathit{fold}\langle 0; acc + 1; id\rangle\{\!\{id\}\!\}(r)$. (Note that we often use the folding constructor together with the aggregate application constructor.) Similarly, the sum of the B components of the tuples in r is given by $\mathit{fold}\langle 0; acc + B(curr); id\rangle\{\!\{id\}\!\}(r)$ or, equivalently, by $\mathit{fold}\langle 0; acc + curr; id\rangle\{\!\{B\}\!\}(r)$. The average of the B components can be computed by evaluating the sum and the count in parallel, and then taking their ratio:

$$\mathit{fold}\langle [S = 0, C = 0]; [S = acc.S + curr.B, C = acc.C + 1]; S/C\rangle\{\!\{id\}\!\}(r).$$

Folding also allows computing aggregations over nested sets. For instance, let s be an expression of type $\{[A : \mathcal{D}, B : \{\mathbb{Q}\}]\}$, and assume that we want to extend each tuple by a new component holding the sum of the set of numbers occurring in the B component. This can be obtained by:

$$\mathit{replace}\langle [A, B, C = \mathit{fold}\langle 0; acc + curr; id\rangle\{\!\{id\}\!\}(B)]\rangle(s).$$

This expression suggests that an SQL query with the *group-by* clause can be specified in this algebra by applying *fold* after a nesting.

The restriction on the type τ_a imposed in the definition of folding signature has been introduced to guarantee tractability of the folding constructor. In fact, the *unrestricted* folding constructor, fold^{unr}, in which the accumulator can be of any complex type, can express very complex data manipulations, including *replace*, *set-collapse*, and *powerset*:

$$\mathit{powerset}(s) = \mathit{fold}^{unr}\langle\{\{\}\}; acc \cup \mathit{replace}\langle id \cup \{curr\}\rangle(acc); id\rangle\{\!\{id\}\!\}(s).$$

[3] A binary function f is *left-commutative* if it satisfies the condition $f(x_1, f(x_2, y)) = f(x_2, f(x_1, y))$. Commutativity implies left-commutativity, but not vice versa.

4.2 Numeric Folding

We now consider a further constraint on the definition of folding, which essentially allows only the computation of *numeric* aggregations. This is coherent with our approach that tends to make a clear distinction between the restructuring capabilities of a query language and its ability to compute aggregations.

Definition 5. *Let Ω be a vocabulary. A* numeric folding over Ω *is a folding constructor ϕ that satisfies the following conditions:*

- *the signature of ϕ has the form $(\mathbb{Q}, \tau_a, \mathbb{Q})$, where τ_a is either $\mathbb{Q}$ or $[A_1 : \mathbb{Q}, \ldots, A_k : \mathbb{Q}]$, with $k > 0$;*
- *the pre-function, the increment function, and the post-function of ϕ can use only the following functions and constructors: the identity function, binary composition, the labeled tuple constructor, the scalar functions in Ω, the numeric folding constructor, and possibly the attribute names $A_1, \ldots, A_k$.*

The notion of numeric folding is important since it is akin to User-Defined Aggregates [12] in SQL3, that is, the way in which existing commercial database systems allow the user to define its own aggregate functions.

Definition 6. *A* counting folding *over a vocabulary Ω is a numeric folding over Ω in which the increment function does not use the symbol curr.*

Lemma 2. *Let Ω be a vocabulary. Then:*

- *every numeric folding over Ω computes a uniform aggregate function over Ω;*
- *every counting folding over Ω computes a uniform counting function over Ω.*

For instance, $\mathit{fold}\langle 0; \mathit{acc} + \mathit{curr}; \mathit{id}\rangle$ is a numeric folding that computes the aggregate function SUM, whereas $\mathit{fold}\langle 0; \mathit{acc} + 1; \mathit{id}\rangle$ is a counting folding that computes the counting function COUNT.

4.3 Expressive Power of Query Languages with Folding

In this section we relate the language $\mathcal{CVA}+\Omega+\mathit{fold}$, in which aggregations are computed using the folding constructor, with $\mathcal{CVA}+\Omega+\overline{\mathit{fold}}$, in which only numeric foldings are allowed. We consider also the weaker languages $\mathcal{CVA}+\Omega+\mathit{fold}^c$ and $\mathcal{CVA}+\Omega+\overline{\mathit{fold}}^c$, in which the former disallows the use of the symbol *curr* (and thus increment functions cannot refer to the current element while iterating over a collection) and the latter has only the counting folding constructor.

It is clear that (syntactically) $\mathcal{CVA}+\Omega+\overline{\mathit{fold}} \subseteq \mathcal{CVA}+\Omega+\mathit{fold}$, $\mathcal{CVA}+\Omega+\overline{\mathit{fold}}^c \subseteq \mathcal{CVA}+\Omega+\mathit{fold}^c$, and $\mathcal{CVA}+\Omega+\overline{\mathit{fold}}^c \subseteq \mathcal{CVA}+\Omega+\overline{\mathit{fold}}$. For each of these containments ($\subseteq$) it is interesting to ask whether it is strict ($\subset$) or if the syntatic restrictions is ineffective ($=$).

As an example, consider the following $\mathcal{CVA}+\Omega+\mathit{fold}$ expression over the complex value s of type $\{\{\mathcal{D}\}\}$ (a set of sets) that computes the sum of the cardinalities of the sets in s:

$$\mathit{fold}\langle 0; \mathit{acc} + \mathit{fold}\langle 0; \mathit{acc}+1; \mathit{id}\rangle \{\!\{\mathit{id}\}\!\}(\mathit{curr}); \mathit{id}\rangle \{\!\{\mathit{id}\}\!\}(s).$$

The foregoing expression does not belong to $\mathcal{CVA}+\Omega+\overline{\mathit{fold}}$, because the outer folding is not numeric (since it applies to a set of sets). There exists however an equivalent $\mathcal{CVA}+\Omega+\overline{\mathit{fold}}$ expression:

$$\mathit{fold}\langle 0; \mathit{acc} + \mathit{curr}; \mathit{id}\rangle \{\!\{\mathit{fold}\langle 0; \mathit{acc}+1; \mathit{id}\rangle \{\!\{0\}\!\}\}\!\}(s).$$

This example suggests that sometimes expressions involving generic foldings can be rewritten into expressions involving only numeric foldings, by exploiting the restructuring capabilities of $\mathcal{CVA}$. This is confirmed by the following result.

Theorem 2. *Let Ω be a vocabulary. Then:*

- *$\mathcal{CVA} + \Omega + \mathit{fold}$ and $\mathcal{CVA} + \Omega + \overline{\mathit{fold}}$ have the same expressive power;*
- *$\mathcal{CVA} + \Omega + \mathit{fold}^c$ and $\mathcal{CVA} + \Omega + \overline{\mathit{fold}}^c$ have the same expressive power.*

This result states that $\mathcal{CVA}$ embedding the class of aggregate functions that can be computed by numeric foldings over a given vocabulary is essentially equivalent to $\mathcal{CVA}$ with the folding operator.

Modern commercial database systems allow the user to define its own aggregate functions in a way similar to numeric folding. Theorem 2 shows that these systems have taken a right choice, since the introduction of a more general folding operator would not have added expressive power.

5 Querying with Aggregate Functions

In this section we study the relationship between the class of aggregate functions that can be expressed by the numeric folding and the class of the uniform aggregate functions, as well as the effect of their inclusion in $\mathcal{CVA}$.

We first show that numeric folding, considered as a stand-alone operator, is weaker than uniform aggregation. Specifically, the following result states that any aggregate function expressed by a numeric folding can be described by a uniform aggregate function whose description has size linear in the number of its arguments.

Let $\mathcal{F}(\Omega)$ be the class of aggregate functions that can be computed by numeric foldings over a vocabulary Ω, and $\mathcal{F}^c(\Omega)$ be the class of counting functions defined by counting foldings over Ω. Moreover, let $\mathcal{C}(\Omega)$ be the class of uniform counting functions over Ω.

Theorem 3. *Let Ω be a vocabulary. Then $\mathcal{F}(\Omega) \subseteq \mathcal{A}^1(\Omega)$ and $\mathcal{F}^c(\Omega) \subseteq \mathcal{C}^1(\Omega)$.*

Actually, we conjecture that $\mathcal{F}(\Omega) = \mathcal{A}^1(\Omega)$ and $\mathcal{F}^c(\Omega) = \mathcal{C}^1(\Omega)$, but we do not have a proof at the moment.

Thus, in general, we have that $\mathcal{F}(\Omega) \subseteq \mathcal{A}(\Omega)$ and $\mathcal{F}^c(\Omega) \subseteq \mathcal{C}(\Omega)$ but, according to Lemma 1, there are vocabularies for which the containment is proper, that is, $\mathcal{F}(\Omega) \subset \mathcal{A}(\Omega)$ and $\mathcal{F}^c(\Omega) \subset \mathcal{C}(\Omega)$. These containments suggest that numeric folding presents a limitation in computing aggregate functions, as it is not able to capture an entire class $\mathcal{A}(\Omega)$. On the other hand, it is possible to show that this deficiency can be *partially* remedied by the restructuring capabilities of a query language. For instance, consider again the uniform aggregate function α introduced in the proof of Lemma 1, with $\alpha_n = \bigoplus_{i,j} x_i \otimes x_j$, which belongs to $\mathcal{A}^2(\Omega_{abs}) - \mathcal{A}^1(\Omega_{abs})$ and, as such, it cannot be expressed as an aggregate *function* by a numeric folding. However, it can be expressed as an aggregate *query* in $\mathcal{CVA} + \Omega_{abs} + \overline{\mathit{fold}}$, as follows:

$$\mathit{fold}\langle 0_\oplus; \mathit{acc} \oplus \mathit{curr}; \mathit{id}\rangle \{\!\!\{ A_1 \otimes A_2 \}\!\!\} (\mathit{cross}_{[A_1,A_2]}(\mathit{id}, \mathit{id})).$$

Lemma 3. *Let Ω be a vocabulary. Then, for each $k > 0$:*

- *there are functions in $\mathcal{A}^k(\Omega) - \mathcal{A}^{k-1}(\Omega)$ that can be expressed in $\mathcal{CVA} + \Omega + \overline{\mathit{fold}}$;*

– *there are functions in* $\mathcal{C}^k(\Omega) - \mathcal{C}^{k-1}(\Omega)$ *that can be expressed in* $\mathcal{CVA} + \Omega + \overline{fold}^c$.

Actually, there are uniform aggregate functions that cannot be expressed using aggregate queries with folding, showing that, in general, it can be better to have at disposal a language for expressing aggregate functions (such as the uniform ones) outside the database query language.

Theorem 4. *There exists a vocabulary* Ω *such that:*

– $\mathcal{CVA} + \Omega + \mathcal{A}(\Omega)$ *is more expressive than* $\mathcal{CVA} + \Omega + \overline{fold}$*;*
– $\mathcal{CVA} + \Omega + \mathcal{C}(\Omega)$ *is more expressive than* $\mathcal{CVA} + \Omega + \overline{fold}^c$.

Sketch of proof: Consider again the abstract vocabulary Ω_{abs} introduced in the proof of Lemma 1, and assume that the binary function $\otimes$ is commutative. Then, the family γ of functions such that $\gamma_n = \bigoplus_{i,j}^{i<j} x_i \otimes x_j$ represents a uniform aggregate function, which belongs to $\mathcal{A}^2(\Omega_{abs}) - \mathcal{A}^1(\Omega_{abs})$. It turns out that γ cannot be expressed using $\mathcal{CVA} + \Omega_{abs} + \overline{fold}$. $\square$

6 Conclusions and Future Work

We believe that the framework proposed in this paper can be fruitfully used as a formal foundation for further studies on the relationship between aggregate functions and aggregate queries. In particular, the relationship between the classes $\mathcal{CVA} + \Omega + \overline{fold}$ and $\mathcal{CVA} + \Omega + \mathcal{A}(\Omega)$ deserves a deeper investigation.

On one hand, we plan to investigate the implications on considering mutual properties of functions in a vocabulary, such as commutativity, associativity, and distributivity. For instance, the function γ introduced in the proof of Theorem 4 is an aggregate function only if $\otimes$ is commutative. There are conditions (over a vocabulary Ω) that imply that $\mathcal{F}(\Omega) = \mathcal{A}^1(\Omega)$ or that $\mathcal{CVA} + \Omega + \overline{fold}$ and $\mathcal{CVA} + \Omega + \mathcal{A}(\Omega)$ have the same expressive power?

On the other hand, there are simple *logspace* computations that cannot be easily captured by folding. The definition of the functions β (Remark 1) and γ (proof of Theorem 4) shows that uniform construction can compare indexes of the arguments (as $i < j$ in $\bigoplus_{i,j}^{i<j} x_i \otimes x_j$ and $i \neq j$ in $\bigoplus_{i,j}^{i\neq j} x_i \otimes x_j$). Such a capability can be partially captured by referring to total ordered domains (both $\mathcal{D}$ and $\mathbb{Q}$) and a total order predicate $\geq$ (both as a base function in $\mathcal{CVA}$ and in the vocabulary Ω). This extension is another topic that we plan to investigate, first of all by tackling the following claim:

Conjecture 1. Let Ω be a vocabulary. Then $\mathcal{CVA} + \Omega + \overline{fold}$ and $\mathcal{CVA} + \Omega + \mathcal{A}(\Omega)$ have the same expressive power over totally ordered domains.

References

1. S. Abiteboul and C. Beeri. The power of languages for the manipulation of complex values. *The VLDB Journal*, 4(4):727–794, October 1995.
2. M. Benedikt and H. J. Keisler. Expressive power of unary counters. In *International Conference on Data Base Theory*, Springer–Verlag, pages 291–305, 1997.
3. M. Benedikt and L. Libkin. Languages for relational databases over interpreted structures. In *Sixteenth ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems*, pages 87–98, 1997.

4. L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bull. Amer. Math. Soc.*, 21:1–46, 1989.
5. P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.
6. L. S. Colby. A recursive algebra for nested relations. *Information Systems*, 15(5):567–582, 1990.
7. M. P. Consens and A. O. Mendelzon. Low complexity aggregation in GraphLog and Datalog. In *International Conference on Data Base Theory*, Springer–Verlag, pages 379–394, 1990.
8. K. Etessami. Counting quantifiers, successor relations, and logarithmic space. *Journal of Computer and Systems Science*, 54(3): 400–411, 1997.
9. E. Grädel and Y. Gurevich. Metafinite model theory. In *Logic and Computational Complexity*, Springer–Verlag, pages 313–366, 1995.
10. E. Grädel and K. Meer. Descriptive complexity theory over the real numbers. In *Twenty-Seventh Annual ACM Symposium on Theory of Computing*, pages 315–324, 1995.
11. S. Grumbach and C. Tollu. On the expressive power of counting. *Theoretical Computer Science*, 149(1):67–99, 1995.
12. ISO and ANSI. Database Language SQL3 (ISO/ANSI Working Draft). ISO DBL LHR-004, ANSI X3H2-95-364, 1995.
13. R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 869–941. Elsevier Science Publishers (North-Holland), Amsterdam, 1990.
14. A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.
15. L. Libkin and L. Wong. New techniques for studying set languages, bag languages and aggregate functions. In *Thirteenth ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems*, pages 155–166, 1994.
16. L. Libkin and L. Wong. Query languages for bags and aggregate functions. *Journal of Computer and System Sciences*, 55(2):241–272, 1997.
17. G. Özsoyoglu, Z. M. Özsoyoglu, 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.
18. A. Poulovassilis and C. Small. Algebraic query optimisation for database programming languages. *The VLDB Journal*, 5(2):119–132, April 1996.
19. W. L. Ruzzo. On uniform circuit complexity. *Journal of Computer and Systems Science*, 22: 365–383, 1981.
20. H. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.
21. S. J. Thomas and P. C. Fisher. Nested relational structures. In *Advances in Computing Research: the theory of database*, JAI Press, 1986.