Proc. Int. Workshop on Database Programming Languages (DBPL), 1989

# On Bulk Data Type Constructors and Manipulation Primitives: A Framework for Analyzing Expressive Power and Complexity[1]

Richard Hull and Jianwen Su
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782
USA
Hull@cse.usc.edu JSu@cse.usc.edu

**Abstract**

We present a framework for analyzing the complexity and expressive power of many existing languages for manipulating information in bulk data types. The framework is based on four dimensions: the "cardinality measure" of the underlying data model; procedural vs. declarative; the presence of iteration; and the presence of "invention". Several languages from the literature which correspond to various combinations of these dimensions are categorized into five families of queries, ranging from QLOGSPACE to a family which is more expressive than the computable queries.

## 1 Introduction

The integration of database technology with programming languages is emerging as an important topic for the 1990s [AB87]. A central problem concerns how the 'bulk' data types of typical database applications can be combined with the powerful operators common to most programming languages. Recent theoretical results imply that some naive combinations of database and programming language capabilities result in manipulation languages which permit highly intractable — and in some cases noncomputable — programs. This paper provides a framework for studying the interaction between data constructs and manipulation primitives from these two areas, and uses it to provide a perspective on the growing body of theoretical results in this realm.

In the context of database programming languages it is appropriate to distinguish between *bulk data* and *computational data*. While the boundary between these two kinds of data is by no means sharp, the phrase 'bulk data' refers in general to the persistent data manipulated by an application which in a conventional environment would be managed by a database, while 'computational data' refers to other, typically more transient data arising in the application. In general, bulk data is characterized by large quantities of data with fairly uniform structure, and computational data is typically of smaller quantity but much more varied structure.

In this paper we focus on a particular kind of manipulation of bulk data, here called *generic manipulation*. This is the kind of manipulation which treats data values in an essentially uniform or "uninterpreted" manner, and arises in such applications as conventional database query languages, report generation facilities, database sharing and restructuring activities, etc. (A number of researchers have isolated the property of

[1] This work supported in part by NSF grant IRI-87-19875 and a grant from AT&T; the first author was also supported in part by DARPA contract MDA903-81-C-0335. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official opinion or policy of NSF, AT&T, DARPA, the U.S. Government, or any other person or agency connected with them.

genericity as a key element of database query languages [AU79, CH80, HY84]; briefly, a mapping is *generic* if it commutes with permutations of the underlying domain of values which leave a finite set of values fixed.) In particular, in most of this paper we do not consider computations based on atomic values (e.g., arithmetic, string manipulation, or other such functions), and instead focus primarily on combinations of primitive generic operations associated with the fundamental bulk data type constructs.

In order to study and compare the complexity and expressive power of proposed bulk data type constructors and manipulation primitives, we focus on mappings from flat relations to flat relations. In Section 2 we describe five classes of such mappings, based on complexity and expressive power. These range from the relational calculus and algebra, which have logarithmic space complexity, to languages which express noncomputable queries.

In Section 3 we focus on the data constructs provided by a variety of database models and database programming languages. In the context of complexity and expressive power, there is a natural categorization of models of bulk data representation into three classes. This categorization stems from the *data* (or *information*) *capacity* [HY84, Hul86] (or representational power) of the types and/or schemas in the data models. Representatives of these classes are the relational model (based on tuples), the complex object model (which includes tuples and nested sets), and the model of GALILEO (which includes tuples and sequences, and can simulate nested sets), respectively. In this section we categorize some 24 data models according to this classification.

The various data constructs discussed in Section 3 have natural manipulation primitives associated with them. In Section 4 we focus on higher level mechanisms for combining these construct-specific manipulation primitives. In particular, we propose a three-dimensional taxonomy of such mechanisms. One dimension focuses on the dichotomy between the procedural and declarative approaches, and another focuses on the presence or absence of iteration (e.g., while-loops, fixpoints, etc.). The final dimension focuses on *invention*, originally introduced in [AV87]. Analogies of invention arise in several practical languages, e.g., the **new** construct of object-oriented languages, and through "external" functions (e.g., arithmetic operations, string concatenation, etc.).

In Section 5 we characterize the complexity and expressive power (using the classes of Section 2) of a variety of combinations of constructs and mechanisms from Sections 3 and 4. The characterizations are based on known results for specific languages; these results are cited in the Appendix. The survey presented in Section 5 is not exhaustive, but rather indicates the spirit of known results. Brief concluding remarks are presented in Section 6.

The focus of this paper is primarily philosophic. The definitions and stated results are presented in an informal fashion in order that they be more easily accessible to a broad audience.

# 2 Classes of Complexity and Expressivity

In this section, we first briefly introduce some basic notions including *query functions*; and then define five families of them in terms of their complexity and expressive power. As mentioned in the introduction, our languages encompass different data models; we consider here only query functions whose domains are relational databases and ranges are flat relations, i.e., queries on relational databases. This common basis allows us to compare how different bulk data types and control primitives affect expressive power and query complexity.

Because our interest is in results of a fairly coarse nature, we also make some simplifying assumptions. We assume the existence of two disjoint and countably infinite sets of *predicate names* and *attribute names* (respectively); and the existence of a set of *basic* types (integer, boolean, character, string, ...) with associated domains (disjoint, countable). At this stage we also disregard the possible presence of object identity, as arises in many semantic and object-oriented models. (The only substantial impact of object identity arises if the **new** operator is used; this will be considered in our discussion of 'invention' given below.)

**Definition:** A *relation schema* $T$ consists in a *relation name*, which is a predicate name, and a finite set of pairs, each of which contains an attribute name and a basic type. A (*relation*) *instance* of $T$ is a pair consisting of its relation name and a finite set of tuples of elements from the corresponding domains. A (*relational* or *flat*) *database schema* $D$ is a finite set of relation schemas. A (*database*) *instance* of $D$ is a set of corresponding relations.

As is customary in database theory, in this paper we generally restrict our attention to bulk data manipulation which is 'generic' in the following sense:

**Definition:** Let $\mathbf{U}$ be the set of all atomic elements of basic types. A mapping (on database instances) is *generic* if there is some finite subset $C$ of $\mathbf{U}$ such that for each permutation $\sigma$ on $\mathbf{U}$ which respects the basic types and leaves $C$ fixed, $\sigma \circ f = f \circ \sigma$ (where $\sigma$ is extended to database instances in the natural manner). If an ordering of domain elements is assumed, then only permutations $\sigma$ which respect that ordering are considered.

Essentially this notion was introduced independently in [AU79, CH80, HY84] as a natural property for database mappings, because it enforces the intuition that (almost) all elements of the underlying domain are treated in a uniform manner. This assumption has been followed in most theoretical investigations of database query and transformation languages, but may be less appropriate in the context of database programming languages, where more structure for the underlying domain of atomic values is known. While dropping the assumption of genericity made here does not materially affect the nature of our framework, it may affect some aspects of the theory, and would probably yield a more accurate perspective on specific models and languages of interest. In all cases, dropping the assumption of genericity might increase the complexity or expressivity of a language, but will never decrease it.

The following definition of *query functions* reflects our focus on relational queries.

**Definition:** If $D$ is a flat database schema and $T$ is a relation schema, a *query function* $f$ from $D$ to $T$, denoted $f : D \to T$, is a (partial) generic mapping from instances of $D$ to instances of $T$.

Informally, a relational query language $L$ is a set of query expressions with certain semantics, under which each query expression *realizes* (or *expresses*) some query function. We extend this to access languages on arbitrary models by focusing on those queries which map from (simulations of) flat database schemas to (simulations of) relation schemas. Hence, the expressive power and/or complexity of an access language $L$ can be characterized by the set of query functions it realizes. The following notion due to [CH82, Var82] provides the basis for these characterizations.

**Definition:** If $f$ is a query function and $d$ a database, the (data) complexity of $f$ is the complexity of its graph $\{(o, d) \mid o \in f(d)\}$. A set of query functions $F$ (similarly, a language $L$) is said to be *data-complete* in a complexity class $C$ if $F$ is contained in $C$ and every set in $C$ is logspace reducible to some $f \in F$.

Intuitively, the complexity of a query is based on the difficulty of determining, given an input instance and possible output tuple, whether that tuple is produced by the query. This is not an especially realistic measure of the difficulty of computing query answers, because databases typically generate answer tuples, not simply verify them. However, as observed in [CH82], if only generic queries are considered then the total space of possible answers is polynomial in the size of the input. The complexity of generating the set of answer tuples is thus at most a polynomial multiple of verifying a given answer tuple. Furthermore, this measure has the advantage of viewing query complexity as a decision problem, thus making it possible to use both deterministic and nondeterministic automata in characterizations. A provocative open problem is to develop and study more realistic measures of query complexity, especially at the lowest part of the spectrum which lies within PTIME.

The table in Figure 1 shows the five families of queries, their corresponding complexity classes, and some example languages. In the figure, LOGSPACE (PSPACE, HYP-EXP) denote the class of sets recognizable by Turing machines in logspace (polynomial space, hyper-exponential[2] time or space). Also, "computable"

[2] The class of *hyper-exponential* functions $hyp_i$ ($i \in \mathbf{N}$) is defined such that $hyp_0(n) = n$ and $hyp_{i+1}(n) = 2^{hyp_i(n)}$, for $i \geq 0$.

| Family | Complexity | Examples |
|---|---|---|
| QLOGSPACE | LOGSPACE | relational algebra*, relational calculus* |
| QPSPACE | PSPACE | relational algebra + while (assuming $\leq$) |
| $\mathcal{E}$(lementary) | HYP-EXP | complex object algebra, complex object calculus |
| $\mathcal{C}$(omputable) | computable | QL, GALILEO |
| $\mathcal{A}$(rithmetical) | arithmetical | complex object calculus with countable invention, "pure" OPAL in Gemstone |

* in terms only of complexity, not expressive power

Figure 1: Five families and their example languages

means computable by Turing machines and "arithmetical" means lying within the arithmetical hierarchy [Rog87] (or equivalently, computable by Turing machines using other Turing machines as oracles (recursively)). Here family names QLOGSPACE, QPSPACE, $\mathcal{E}$, etc. denote the families of all query functions in the associated complexity classes.

In Figure 1 the relational calculus and algebra are data-complete in LOGSPACE. The remaining four rows have exact correspondences in both expressive power and complexity. The relational algebra + **while** with an assumed order on domain elements realizes exactly the set of all query functions in PSPACE [Var82]; an analogous characterization of the expressive power of the algebra + **while** without order remains open.[3] The complex object algebra and calculus (which will be introduced briefly in Section 3) have expressive power $\mathcal{E}$ [HS88b] (see also [KV88]).

The notion of computable queries was first introduced with the language QL [CH80], and additional computable languages TL [AV87], detTL, and DL [AV88] have been introduced. The GALILEO language, and some other implemented database programming languages, also have the expressive power of $\mathcal{C}$. To some extent this is true because these languages permit the free mixing of operations on bulk and computational data. As will be seen below, however, even if the computational operations are ignored, these implemented languages still yield the computable queries.

The complex object calculus with countable invention [HS88b] has the same expressive power as the arithmetical hierarchy. Because this includes queries which are not computable (by Turing machines), this class is primarily of theoretical interest. The OPAL language of Gemstone [CM84] supports conventional object-oriented methods, and also some declarative constructs such as SELECT (which can be nested) [Ull88]. Using the "pure", i.e., declarative, semantics for the SELECT construct, OPAL has the expressive power of the complex object calculus with countable invention, i.e., of $\mathcal{A}$. Of course, the implemented version(s) of OPAL cannot exceed the expressive power of the family $\mathcal{C}$.

The five classes of expressive power that we focus on here are intended to provide an overall structure for comparing bulk data manipulation languages. The gap between QLOGSPACE and QPSPACE contains QPTIME, the family of query functions with polynomial time complexity. Although it is not know whether these three classes are equal, quite a few known languages lie in the gap: calculus + positive transitive closure [Imm87], stratified DATALOG, calculus + fixpoint which is equivalent to inflationary DATALOG, algebra + while, etc. (see also [Cha81, Cha88]). Also, two noncollapsing hierarchies have been exhibited in the gap between QPSPACE and $\mathcal{E}$ [HS88b, KV88]; in both cases the hierarchies are based on the level of set nesting used by intermediate types in queries.

[3] The absence of an order relation does not affect typical models and languages found in the richer expressivity categories. This is because in general they can simulate an order, at least for the purpose of performing the simulation of a Turing computation. The first result of this sort is found in [Fag74], which shows that the existential second-order relational queries have the expressive power of (generic) NPTIME.

# 3 A Taxonomy of Database Models based on Data Capacity

The constructs suggested by database programming languages for representing bulk data types include most central constructs from existing database models and programming languages. Foremost among these are *tuple* (or *record*), *set*, and *sequence* (or *list*). Different models permit different combinations of these and other constructs; for example a database from the relational model is essentially a tuple of sets of tuples, while GALILEO permits bulk data types constructed using arbitrary nesting of these three constructs.[4] In this section we establish three categories of data models used in database programming languages for representing bulk data. These categories stem in part from the notion of relative data capacity [HY84, Hul86], and focus on how much data the types from the model can "hold". As will be seen below, the categorization is extremely relevant to the analysis of the expressive power of bulk data type manipulation languages (at least in the case where invention is not permitted).

To set the stage for this discussion, we first introduce a representative model for each of the three categories of data models. The weakest category of models is called *polynomial*, and is represented by the relational model [Ull88]. The middle category is called *hyper-exponential*, and is represented by the Complex Object model [Hul87]. Data types in this model are constructed using (possibly nested) tuple and set constructs; the Complex Object model can be viewed as a slight generalization of the nested relation model [FT83, JS82]. The strongest category is called *unbounded*, and is represented here by GALILEO [ACO85], a full-fledged database programming language which supports explicit constructs for tuples, variant records, and sequences.

The choice of names for the categories is based on the size of database instances that can be built from a fixed set of atomic elements. To make this more precise, we introduce a few definitions.

**Definition:** Let $T$ be a bulk data type. If $I$ is an instance of $T$, then the *active domain* of $I$, denoted $adom(I)$, is the set of atomic elements which occur in $I$. If $X$ is a set of atomic elements, then the *constructive domain* of $T$ using $X$ is

$$cons_T(X) = \{I \mid I \text{ is an instance of } T \text{ and } adom(I) \subseteq X\}.$$

**Definition:** Let $T$ be a bulk data type. The *cardinality measure* of $T$ is the function $f_T : \mathbf{N} \to \mathbf{N} \cup \{\omega\}$ defined by[5]

$$f_T(n) = \max\{\|I\| \mid I \in cons_T(X)\}$$

where $X$ is some (any) set of atomic elements with cardinality $n$.

In general, if $T$ is a type and $X$ is a set of atomic elements, then[6] $|cons_T(X)|$ usually has size $\Theta(2^{f_T(|X|)})$.

Suppose now that $T$ is a relational schema. It is easily verified that the cardinality measure $f_T$ of $T$ is a polynomial function whose degree is the maximum arity of relations in $T$. For a complex object type $T$, $f_T$ is bounded by a hyper-exponential function [HY84]. Also, each hyper-exponential function is exceeded by $f_T$ for some complex object $T$. Finally, if $T$ is a type from GALILEO which involves a(n unconstrained) sequence of atomic elements, then $f_T(n) = \omega$ for each $n \geq 1$; for this reason the data model of GALILEO is categorized as unbounded.

The significance of the cardinality measure stems from the fact that many data manipulation languages, primarily those from databases, do not permit the "invention" or use of new atomic values during a computation. The same property holds for some forms of generic manipulation of bulk data types in database programming languages. In all of these cases, the number of possible values of a variable $v$ of type $T$ used in a computation is bounded by $|cons_T(X)|$ where $X$ is (essentially) the active domain of the input; and

[4] Technically, set is not directly supported in GALILEO; it can be simulated using sequences and some constraints.

[5] For an instance $I$, $\|I\|$ denotes the amount of space needed to write $I$ (according to some established conventions, and assuming that each atomic element takes one unit of space).

[6] For a set $Y$, $|Y|$ denotes the cardinality of $Y$.

| *polynomial* | *hyper-exponential* | *unbounded* |
|---|---|---|
| Entity-Relationship | COL | AP5 |
| FDM | Complex Object | Database Logic |
| Nested Relations in | (typed) FAD | (untyped) FAD |
| Partition Normal Form | Format Model | GALILEO |
| Relational | IFO | Gemstone |
| V-relational (VERSO) | LDM queries | LDL |
| | Nested Relation | LDM data model |
| | SDM | $O_2$ |
| | | ORION |
| | | Unranked Relational |
| | | VBASE |

Figure 2: Categorization of data models based on cardinality measure function

the data content of each of these values is intuitively bounded by $f_T(n)$, where $n = |X|$. In the cases of the polynomial and hyper-exponential models, this results in strict upper bounds in the amount of working space or "scratch paper" that the data manipulation can use.

Figure 2 shows where some 24 models fit within the categorization presented here. We now briefly justify our placement of the models in their respective categories. Consider first the polynomial models. The original Entity-Relationship Model [Che76] (and most of its generalizations [BLN86]) does not directly support sets; it can only simulate them with relationships. Sets in FDM [Shi81] can arise only in the ranges of multi-valued attributes. In the V-relational model [AB86], or equivalently, Nested Relations in Partition Normal Form [RKS88], each nested relation must satisfy certain key dependencies. As a result, each nested relation is equivalent to a flat relation obtained from repeated unnesting. Therefore, the number of sets present in an instance is bounded by the number of atomic elements present in the instance; the space to write all these sets is thus polynomially bounded.

It was observed above that the Complex Object and Nested Relation models lie within the hyper-exponential category. The data models of the Format Model [HY84], IFO [AH87a], COL [AG88], and the Semantic Data Model (SDM) [HM81] include the full power of the Complex Object model, along with features such as attributes and ISA relationships which do not affect their cardinality measures. In typed FAD [DV88], the type constructors, among others, include tuple and (homogeneous) set and in particular the model supports complex objects. The inclusion of LDM [KV84] queries in this category is discussed at the end of this section.

As noted above, the presence of the sequence construct in GALILEO places it in the unbounded category. The AP5 [Coh86], LDL [BNR+87], ORION [BCG+87], and VBASE [AH87b], models all include the sequence construct, and are thus also unbounded in this sense. Perhaps the first published work on query languages for unbounded models is [CH80], which introduced the *Unranked Relational Model*. In this model, a variable is permitted to hold relations of arbitrarily large width. [CH80] was the first to introduce the notion of *computable queries*, and exhibited a language based on the Unranked Relational Model which can express all computable queries. Two of the remaining models in the list (untyped FAD [BBKV87], Gemstone [CM84]) include types for *untyped* or *heterogeneous sets*, which can hold both tuples and sets which are nested to an arbitrarily deep level. The following example indicates the implications of supporting such types in a data model.

**Example 3.1:** Consider the data model originally introduced for the data manipulation language FAD. (Similar models are used in LDL and an early version of $O_2$ [LRV88], and Gemstone permits these types in instance variables.) In that model, speaking informally, an *object* is defined to be either (a) an atomic object; (b) a tuple constructed from other objects; or (c) a set containing other objects. (In this definition we are suppressing the object identifiers used in FAD.) Thus, a fixed variable ranging over FAD tuple objects can

hold any of the following values:

```
[head:1],
[head:1,tail:[head:1]],
[head:1,tail:[head:1,tail:[head:1]]],
    ⋮
```

and a variable ranging over FAD set objects can hold any of:

$$\{1\}, \quad \{1,\{1\}\}, \quad \{1,\{1,\{1\}\}\}, \quad \cdots$$

It is thus clear that if $T$ is the type of FAD objects, then $f_T(n) = \omega$ for each $n \geq 1$. A theoretical study of the expressive power languages using untyped *sets* was initiated in [HS89b]. □

The final two unbounded models are the data models of Jacobs' Database Logic [Jac82] and of LDM [KV84]. In Database Logic schemas are describe using a context-free grammar formalism reminiscent of that sometimes used for the Nested Relation Model. Unlike nested relations, however, Database Logic permits cycles in the specification of a schema. As a result, a fixed Database Logic schema can hold instances having arbitrarily deep tuple or set nesting.

LDM schemas are defined as directed graphs involving vertices corresponding to a fixed basic type, and to the tuple and set constructs. As with Database Logic, LDM schemas may have cycles. A subtlety here is that LDM distinguishes between $r$-values and $l$-values (see [KV84]). All atomic values occur as $r$-values, and $l$-values serve as a kind of object identifier for complex objects constructed from both $r$- and $l$-values. There are cyclic LDM schemas which have, for each nonempty finite set $X$ of atomic values, an unbounded number of non-isomorphic instances formed from $X$. These different instances use arbitrarily large numbers of $l$-values. Our inclusion of LDM in the unbounded category is motivated by our intuition that the $l$-values used in an instance correspond to an artifact in the representation of LDM instances – they model an aspect of the implementation model rather than the conceptual model.

We now turn to LDM queries. Intuitively, an LDM query (in both the calculus and the algebra of [KV84]) consists in an augmentation of an LDM schema, along with a specification of how each vertex of that augmentation is to be populated. The augmentation must be acyclic, and each of the vertices is visited exactly once in the process of populating them. Because LDM queries support both the tuple and set constructs, they can build query answers which are hyper-exponential in the size of the input instance.

# 4 A Taxonomy of Language Control Mechanisms

Database programming languages accessing bulk data need both (1) construct-specific manipulations and (2) control mechanisms to combine these operations. Here our focus is on some primitive factors at the second level. In particular, we present a three dimensional view of control mechanisms. The three dimensions are: procedural/declarative, invention, and iteration. These are not formally defined, but rather indicate three general aspects of control mechanisms. In some cases refinements of these control mechanisms do affect the expressive power and complexity of data manipulation languages. Also, we do not intend to imply that the three dimensions studied here are complete in any sense, but rather that they provide a useful framework for understanding a large majority of results currently known about data access languages. Other dimensions can also be studied: for example, the recently proposed notion of "nondeterministic" queries of [AV88] introduces another dimension based on determinism vs. nondeterminism.

To be procedural or declarative is an external aspect of languages. Examples of procedural languages are the relational algebra and variants (complex object algebra, LDM algebra, etc.), and most object-oriented

languages. The relational calculus and variants are representatives of declarative languages. Also, there are many hybrids which essentially contain at least one declarative level and one procedural level. These can be further classified into two kinds: (1) procedural outside, e.g., QUEL (unquantified wffs and set operations), stratified DATALOG, COL, LDL; and (2) declarative outside, e.g., "pure" OPAL. As we shall see, declarative and procedural languages are generally equivalent, except with invention or unbounded models.

We now consider the second dimension — invention. Invention consists in the ability for a query to create and use (possibly arbitrarily many) new values which are not present in the input nor in the query. Invention was first introduced in [AV87], where the main purpose was to enlarge the expressive power of a relational language. Using invented values, various languages introduced there (TL) and in [AV88] (detTL and DL) have been shown to express all computable query functions. In [HS88b] the complex object calculus is extended to a semantics which allows the universe to contain a countably infinite set of invented values.

Although the notion of invention was originally used for theoretical study, it has appeared in practice in at least three different forms. First, the operator **new** in object-oriented languages always creates new values (specifically, for new object identifiers). This is an explicit invention. In the same spirit, set construction in LDM or complex object languages has characteristics of invention; but the number of new values is hyper-exponentially bounded. Second if "external" functions (e.g., "+") are present, new values can be obtained through function applications. From a theoretical perspective, the cardinality of the set of potentially invented values is determined by properties of the external function and its range. Finally, in systems that allow data constructs like sequences, lists, or untyped sets, arbitrarily many new values can always be created (in the spirit of Example 3.1) even without an explicit invention mechanism. This forms the basis of a trade-off between invention and unboundedness (see Example 5.3).

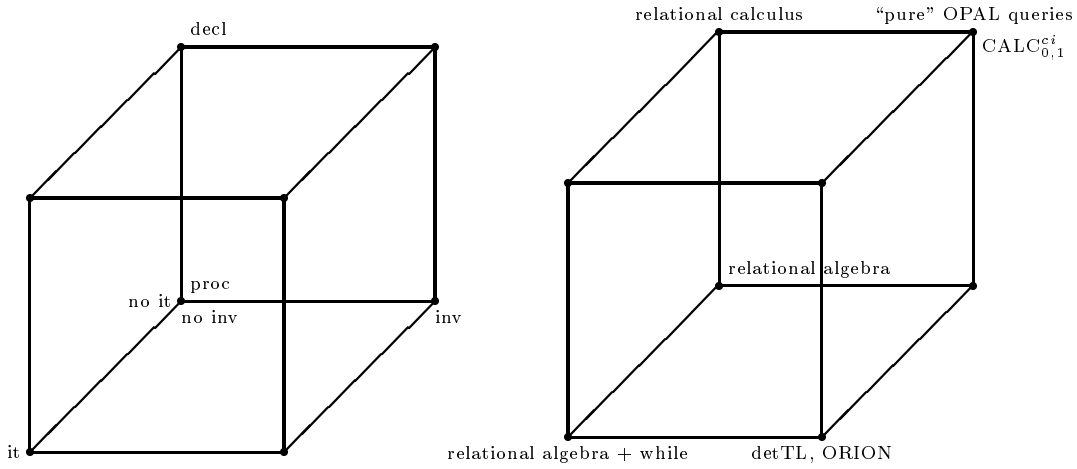
The third dimension, iteration, appears in several forms, in both procedural and declarative languages. In the context of procedural database languages iteration has arisen most frequently in the form of a **while** construct [Cha81]; in some database programming languages it also arises in the form of recursion; and in object-oriented databases it is implicit in the recursive semantics associated with method calls. In the context of declarative languages iteration arises in the form of a fixpoint operator. The best-known examples include the extension of the relational calculus to include fixpoint [CH82] and the use of least fixpoint semantics in deductive databases [CH85]. Of course, the iteration and procedural/declarative dimensions are not truly orthogonal; the incorporation of some forms of iteration into a declarative language can make it essentially procedural. For example, the stratified semantics for DATALOG programs [ABW86], which has also been used for deductive languages for the Complex Object model [AG88, BNR$^+$87], calls for the composition of a sequence of locally applied fixpoints. For this reason we classify the stratified semantics as a procedural/declarative hybrid, which is externally procedural. It is unclear at present how the relational calculus and DATALOG extended by inflationary fixpoint semantics (see [Cha88]) should be classified.

Figure 3(a) shows the three two-element dimensions; each vertex of the cube corresponds to a particular combination of control mechanisms. Note that the choice of models is independent of those three dimensions. In (b), several representative languages are characterized by the three dimensions.

# 5 Classification of Representative Languages

In the previous two sections four dimensions were proposed for categorizing bulk data manipulation languages: the cardinality measure of the underlying data model; declarative vs. procedural; the presence of iteration; and the presence of invention. In this section we indicate where languages based on different combinations of these four factors lie with regards to the hierarchy of complexity and expressivity introduced in Section 2.

The claims of this section are not formal results, and cannot be because the dimensions used have been defined only loosely. The claims made here are based on extrapolations of specific theoretical results which are cited in the Appendix. Importantly, several general assumptions are made about the models and




(a) Three dimensions (b) some representative languages

Figure 3: A three dimensional view

languages characterized in this section. First, as with all models in the literature, it is assumed that all hyper-exponential and unbounded models have data types rich enough to represent Turing computations (e.g., via sets of tuples, sets of sequences, or sequences of sequences, etc.). Also, declarative languages are assumed to have the expressive power of a calculus in the spirit of the relational calculus extended to the models of interest. Thus, for example, we do not include in this study the syntactically restricted complex object calculus proposed in [AB88]. The procedural languages considered are also the richest studied in the literature; e.g., for the hyper-exponential models it is assumed that the powerset operator is included. (As originally shown in [KV84], essentially this operator is needed in order for the complex object algebra to have the same power as the complex object calculus.) Also, we assume that when invention is present, then an arbitrarily large number of new values can be invented; the impact of bounds on this number is briefly studied in [HS88b].

Figure 4 shows the five levels of the expressive power hierarchy, and for each level it shows points in the four-dimensional space which have the associated expressive power. (For example, the tuple [unb, proc, no iter, —] indicates those languages which have unbounded cardinality measure, are procedural, do not have iteration; and may or may not have invention.)

We now briefly illustrate how the framework presented here can be used to provide a variety of insights into the interaction of the four dimensions.

**Example 5.1: Procedural vs. Declarative.** In the absence of iteration or invention, procedural and declarative languages generally have the same expressive power on the polynomial and hyper-exponential models. To some extent this correspondence may be artificial: the equivalence obtained by Codd for the relational case was to some extent imitated in the context of the LDM and the nested relation and complex object models [KV84, RKS88, AB88]. On the other hand, if the powerset operator is omitted from the nested relation algebra, this yields a procedural language which is strictly weaker than the nested relation calculus.

However, in the case of unbounded models calculus-based declarative languages are more powerful than any procedural language can be. In particular, as shown in the chart, these declarative languages fall within the class $\mathcal{A}$, which includes noncomputable queries. Essentially this result was perhaps first observed in

| | |
|---|---|
| QLOGSPACE: | [pol, —, no iter, no inv][1], [pol, decl, no iter, inv][1] |
| | [pol, decl, iter, no inv][2] |
| QPSPACE: | [pol$^{\leq}$, proc, iter, no inv][3] |
| $\mathcal{E}$: | [hyp, —, —, no inv], [unb, proc, no iter, —][4] |
| $\mathcal{C}$: | [unb, proc, iter, —], [—, proc, iter, inv] |
| $\mathcal{A}$: | [unb, decl, —, —], [hyp, decl, —, inv] |

Notes:

1. The categories listed in QLOGSPACE are data complete in LOGSPACE, but have expressive power properly contained in QLOGSPACE.
2. It is known that the relational calculus + fixpoint is stronger than QLOGSPACE, and (assuming $\leq$) is contained in QPSPACE. Whether the latter containment is proper remains open.
3. pol$^{\leq}$ indicates polynomial models in which the atomic domain has a total order.
4. The placement of this combination in $\mathcal{E}$ is based on a result for the complex object model using untyped sets [HS89b]. Other unbounded models, e.g., which include lists but not sets, may not attain this expressive power when combined with the indicated control mechanisms.

Figure 4: Classification of some Categories of Languages

connection with databases in [Var83], which observed that the calculus-based language of Jacobs' Database Logic permits the description of sets which are not recursive. Intuitively, the result holds because in the context of an unbounded model, a universal quantifier can be used to range over all partial computations of a Turing machine [HS88b]. Such a query can, for example, easily decide the halting problem. The expressive power of procedural languages is, of course, bounded by $\mathcal{C}$. □

**Example 5.2: Iteration.** In polynomial models, iteration gives more expressive power. For example, transitive closure cannot be expressed by the relational algebra or calculus [AU79], but can be expressed by relational algebra + **while** and relational calculus + fixpoint. It remains open whether the latter languages have equivalent expressive power, although it is known [Var82] that they do if and only if PTIME = PSPACE, which is generally viewed to be unlikely.

For hyper-exponential models, iteration can be simulated by operations on nested sets [AB88, GvG87], and so does not affect the expressive power of procedural or declarative languages. In particular, a query using iteration and nested sets of depth $k$ can be simulated by a query without iteration using nested sets of depth $k+1$.

In the case of unbounded models, iteration has an impact on procedural languages but not declarative languages. If iteration is included in a procedural language, then the fact that arbitrarily many values are available (for serving as indices of arrays which store encoded computations), allows the simulation of arbitrary Turing computations; thus yielding the expressive power of $\mathcal{C}$. On the other hand, if iteration is not supported, then the procedural language cannot take advantage of unlimited invention, and simulated Turing computations are still space bounded [HS89b].

As noted in the previous example, declarative languages on unbounded models lie in $\mathcal{A}$; incorporating iteration does not increase their expressive power. □

**Example 5.3: Invention vs. Unboundedness.** There is strong correspondence between invention and unboundedness. Procedural languages with iteration on any size model which use invention fall within $\mathcal{C}$ [AV87], as do procedural languages with iteration on unbounded models [HS89b]. In the case of declarative languages, the use of hyper-exponential models with invention and the use of unbounded models both yield the class $\mathcal{A}$ [HS88b, HS89b]. The only point where the trade-off is broken is in the case of declarative

languages on polynomial models (or at least, on the relational model) – adding invention to the relational calculus does not increase its expressive power [HS88a, AGSS86]. □

# 6 Concluding Remarks

The framework presented here provides a perspective on the full range of expressive powers and complexity of practical and theoretical bulk data manipulation languages, ranging from QLOGSPACE to $\mathcal{A}$. As mentioned in Section 2, our categorization is rather coarse, and a considerable amount of structure between the five classes is not emphasized. The influence of programming languages on bulk data manipulation will probably lead to the development of even richer structure in these gaps, because of their different constructs and manipulation primitives.

The framework can be used in the design of database programming languages to get an approximate idea of both the expressive power of proposed bulk data manipulation mechanisms, and the complexity of executing programs using them. In contrast to insisting on rich languages (e.g., equivalent to $\mathcal{C}$), it is conceivable that a spectrum of languages is more suitable for a diversity of users. For example, it may be desirable to restrict the language presented to naive system users to lie within QPTIME or something smaller, while experts are permitted to pose PSPACE or even exponential queries. In the extreme, the DBA might be given the ability to express all computable queries. To achieve these different levels for a given model, different combinations of manipulation primitives can be used. Other techniques, such as syntactic restrictions, can also be applied. For example, syntactically restricted languages have been studied for the Complex Object model [AB88], an object-oriented model [HS89a], and a model supporting object identity [AK89].

The framework presented here may provide motivation to separate some bulk data operations from some computational data operations in the design of database programming languages. While bulk data accesses are restricted, computational data operations may have the full power of Turing machines. As a particular approach, object-orientation provides a natural implementation framework for database programming languages. In the current study, we did not consider methods, inheritance and largely ignored the distinction between object identifiers and values. However, this study of the expressivity and complexity spectrum can be extended to analyze bulk data accessing mechanisms in object-oriented systems. Recently, access languages using object identity and methods have been studied. Abiteboul and Kanellakis studied deductive query languages which use explicit mechanisms to manipulate object identity [AK89]. On the other hand, in [HS89a] access mechanisms are explored in algebraic languages in the presence of inheritance and methods. Both investigations show that languages can be as powerful as $\mathcal{C}$ and both arrive at PTIME sublanguages.

Finally, we briefly mention another neglected but important issue — updates. In the literature, update languages are largely based on transactions [AB76, SWKH76]. Intuitively, there are at least two differences between queries and updates: updates typically access a relatively small portion of a database, and updates must generally preserve integrity constraints. Because of these differences, it seems desirable to develop a new framework for measuring the complexity and expressivity of update languages, perhaps taking into account the size of the affected part of the database. A useful starting point this investigation may be the research reported in [AV85, AV86], which introduces a model for studying transactions and develops results concerning decidability and a kind of expressive power.


# Acknowledgments

The authors benefited from stimulating discussions with Dino Karabeg and Victor Vianu.


# Appendix: Specific formal results

This appendix briefly lists the specific formal results which were used as the basis for the characterizations presented in Figure 4.

[pol, —, no iter, no inv] is placed in QLOGSPACE because both the relational algebra and calculus are known to be data-complete in LOGSPACE [Var82].

[pol, decl, no iter, inv] is placed in QLOGSPACE because the relational calculus with invention is equivalent in expressive power to the relational calculus without invention [HS88a, AGSS86].

[pol, decl, iter, no inv] is shown between QLOGSPACE and QPSPACE, because the relational calculus plus fixpoint is known to be data-complete in PTIME [CH82], and if an order is assumed, to express all PTIME queries [Var82, Imm82].

[pol$^{\leq}$, proc, iter, no inv] is placed in QPSPACE because the relational algebra + **while** is data-complete in PSPACE [Cha81], with an order on the underlying domain it is equivalent to the family of queries in PSPACE [Var82].

[hyp, —, —, no inv] is placed in $\mathcal{E}$ because the complex object languages (algebra, calculus, co-DATALOG) with or without iteration, are equivalent to the family $\mathcal{E}$ [HS88b] (see also [AB88, KV88]).

[unb, proc, no iter, —] is placed in $\mathcal{E}$ because the natural algebra for the complex object model with untyped sets without iteration is equivalent, when restricted to relational input and output, to the complex object model algebra [HS89b]. This result relies in part on the presence of the powerset operator in the complex object algebra, which allows the creation of intermediate values of hyper-exponential size. If something like powerset is not present (e.g., in a model supporting sequences but not sets) then the resulting language may not attain the full expressive power of $\mathcal{E}$.

[unb, proc, iter, —] is placed in $\mathcal{C}$ because the natural algebra with iteration on the complex object model with untyped sets yields $\mathcal{C}$ [HS89b].

[—, proc, iter, inv] is placed in $\mathcal{C}$ because the language detTL, which works on the relational model, and provides primitives for composition, while, and invention, yields the family $\mathcal{C}$ [AV87]. The language DL also yields $\mathcal{C}$ [AV88]. This language has a syntax borrowed from DATALOG, and a semantics related to that of inflationary fixpoint, and is thus a hybrid which is externally procedural.

[unb, decl, —, —] is placed in $\mathcal{A}$ because the calculus on the complex object model with untyped sets yields $\mathcal{A}$ [HS89b].

[hyp, decl, —, inv] is placed in $\mathcal{A}$ because the calculus on the complex object model with infinite invention yields $\mathcal{A}$ [HS88b, HS89b].

# References


[AB76] M.M. Astrahan and M.W. Blasgen etc. System R: Relational approach to database management. *ACM Trans. on Database Systems*, 1(2):97–137, 1976.

[AB86] S. Abiteboul and N. Bidoit. Non first normal form relations: An algebra allowing data restructuring. *Journal of Computer and System Sciences*, 33(3):361–393, 1986.

[AB87] M.P. Atkinson and O.P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, 1987.

[AB88] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. Technical Report No.846, INRIA, May 1988.

[ABW86] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. Technical report, IBM Thomas J. Waston Research Center, 1986. Also in Proc. of Workshop on Foundations of Deductive Databases and Logic Programming, 1986.

[ACO85] A. Albano, L. Cardelli, and R. Orisini. Galileo: A strongly-typed, interactive conceptual language. *ACM Trans. on Database Systems*, 10(2):230–260, 1985.

[AG88] S. Abiteboul and S. Grumbach. COL: A logic-based language for complex objects. In *Advances in Database Technology - EDBT '88*. Lecture Notes in Computer Science, Springer-Verlag, 1988.

[AGSS86] A.K. Aylamazyan, M.M. Gilula, A.P. Stolboushkin, and G.F. Schwartz. Reduction of the relational model with infinite domain to the case of finite domains. *Proc. of USSR Acad. of Science (Doklady)*, 286(2):308–311, 1986.

[AH87a] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Trans. on Database Systems*, 12(4):525–565, 1987.

[AH87b] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. In *Proc. Conf on OOPSLA*, pages 430–440, 1987.

[AK89] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 159–173, 1989.

[AU79] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 110–120, 1979.

[AV85] S. Abiteboul and V. Vianu. Transaction and integrity. In *Proc. ACM Symp. on Principles of Database Systems*, pages 193–204, 1985.

[AV86] S. Abiteboul and V. Vianu. Deciding properties of transactional schemas. In *Proc. ACM Symp. on Principles of Database Systems*, pages 235–239, 1986.

[AV87] S. Abiteboul and V. Vianu. A transaction language complete for database update and specification. In *Proc. ACM Symp. on Principles of Database Systems*, pages 260–268, 1987.

[AV88] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proc. ACM Symp. on Principles of Database Systems*, pages 240–250, 1988.

[BBKV87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. Int. Conf. on Very Large Databases*, pages 97–105, 1987.

[BCG+87] J. Banerjee, H.-T. Chou, J.F. Garza, W. Kim, D. Woelk, and N. Ballou. Data model issues for object-oriented applications. *ACM Trans. on Office Information Systems*, 5(1):3–26, 1987.

[BLN86] C. Batini, M. Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[BNR+87] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Schmueli, and S. Tsur. Sets and negation in a logic database language (LDL1). In *Proc. ACM Symp. on Principles of Database Systems*, 1987.

[CH80] A.K. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[CH82] A.K. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25(1):99–128, 1982.

[CH85] A.K. Chandra and D. Harel. Horn clauses and generalizations. *Journal of Logic Programming*, 1(1):1–15, 1985.

[Cha81] A.K. Chandra. Programming primitives for database languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 50–62, 1981.

[Cha88] A.K. Chandra. Theory of database queries. In *Proc. ACM Symp. on Principles of Database Systems*, 1988.

[Che76] P.P.-S. Chen. The entity-relationship model — toward a unified view of data. *ACM Trans. on Database Systems*, 1(1):9–36, 1976.

[CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, 1984.

[Coh86] D. Cohen. Programming by specification and annotation. In *Proc. of AAAI*, August 1986.

[DV88] S. Danforth and P. Valduriez. The data model of FAD: A database programming language. Technical Report ACA-ST-059-88, MCC, 1988.

[Fag74] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In R. Karp, editor, *Complexity of Computation*, pages 43–73. SIAM-AMS Proceedings, 1974.

[FT83] P.C. Fischer and S.J. Thomas. Operators for non-first-normal-form relations. In *Proc. IEEE Computer Software Applications Conference*, pages 464–475, 1983.

[GvG87] M. Gyssens and D. van Gucht. The powerset operator as an algebraic tool for understanding least fixpoint semantics in the context of nested relations. Technical Report No.233, Indiana Univ., October 1987.

[HM81] M. Hammer and D. McLeod. Database description with SDM: A semantic database model. *ACM Trans. on Database Systems*, 6(3):351–386, 1981.

[HS88a] R. Hull and J. Su. Domain independence and the relational calculus. Technical Report 88-64, Computer Science Dept, Univ of Southern California, 1988.

[HS88b] R. Hull and J. Su. On the expressive power of database queries with intermediate types. Technical Report 88-53, Computer Science Department, Univ of Southern California, 1988. Invited to special issue of *Journal of Computer and System Sciences*.

[HS89a] R. Hull and J. Su. On accessing object-oriented databases: Expressive power, complexity, and restrictions. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 147–158, 1989.

[HS89b] R. Hull and J. Su. Untyped sets, invention, and computable queries. In *Proc. ACM Symp. on Principles of Database Systems*, 1989.

[Hul86] R. Hull. Relative information capacity of simple relational schemata. *SIAM Journal on Computing*, 15(3):856–886, 1986.

[Hul87] R. Hull. A survey of theoretical research on typed complex database objects. In J. Paredaens, editor, *Databases*, pages 193–256. Academic Press (London), 1987.

[HY84] R. Hull and C.K. Yap. The format model: A theory of database organization. *Journal of the ACM*, 31(3):518–537, 1984.

[Imm82] N. Immerman. Relational queries computable in polynomial time. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 147–152, 1982.

[Imm87] N. Immerman. Languages that capture complexity classes. *SIAM Journal on Computing*, 16(4):760–778, 1987.

[Jac82] B. Jacobs. On database logic. *Journal of the ACM*, 29(2):310–332, 1982.

[JS82] B. Jaeschke and H.J. Schek. Remarks on the algebra of non first normal form relations. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 1982.

[KV84] G.M. Kuper and M.Y. Vardi. A new approach to database logic. In *Proc. ACM Symp. on Principles of Database Systems*, pages 86–96, 1984.

[KV88] G.M. Kuper and M.Y. Vardi. On the complexity of queries in the logical data model. In *Proc. Int. Conf. on Database Theory*, pages 267–280, 1988.

[LRV88] C. Lecluse, P. Richard, and F. Velez. $O^2$: An object-oriented formal data model. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 424–433, Chicago, June 1988.

[RKS88] M.A. Roth, H.F. Korth, and A. Silberschatz. Entended algebra and calculus for nested relational databases. *ACM Trans. on Database Systems*, 13(4):389–417, 1988.

[Rog87] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* MIT Press, Cambridge, Mass., 1987.

[Shi81] D.W. Shipman. The functional data model and the data language DAPLEX. *ACM Trans. on Database Systems*, 6(1):140–173, 1981.

[SWKH76] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Trans. on Database Systems*, 1(3):189–222, 1976.

[Ull88] J.D. Ullman. *Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Var82] M.Y. Vardi. The complexity of relational query languages. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 137–146, 1982.

[Var83] M.Y. Vardi. Review of [Jac82]. *Zentralblatt für Mathematik*, 497.68061, 1983.