Fully Generic Queries: Open Problems and Some Partial Answers

Peer-reviewed author version

SURINX, Dimitri; VAN DEN BUSSCHE, Jan & VIRTEMA, Jonni (2019) Fully Generic Queries: Open Problems and Some Partial Answers. In: Schewe, Klaus-Dieter; Kumar Singh, Neeraj (Ed.). Model and Data Engineering, NATURE PUBLISHING GROUP, p. 20 -31..

DOI: 10.1007/978-3-030-32065-2_2
Handle: http://hdl.handle.net/1942/30375

Fully Generic Queries: Open Problems and Some Partial Answers

Dimitri Surinx, Jan Van den Bussche[0000−0003−0072−3252], and Jonni Virtema[0000−0002−1582−3718]

Hasselt University
jan.vandenbussche@uhasselt.be

Abstract. The class of fully generic queries on complex objects was introduced by Beeri, Milo and Ta-Shma in 1997. Such queries are still relevant as they capture the class of manipulations on nested big data, where output can be generated without a need for looking in detail at, or comparing, the atomic data elements. Unfortunately, the class of fully generic queries is rather poorly understood. We review the big open questions and formulate some partial answers.

1 Introduction

For the task of querying a database, database systems offer a database query language. Unlike a general-purpose programming language, a database query language allows us to formulate queries on the logical level of the data model on which the database system is based. Working on that higher, logical level has many advantages. First, programs can be correct independently of how the data is physically stored. Moreover, it makes it easier for the database query processor to recognize "tractable" parts of queries that can be processed more efficiently. Tractability here can mean many things: a selection on an attribute for which an index is available; a join operation for which a specific algorithm can be used; and so on [17].

Historically, the logical nature of database queries was explicated independently by several researchers [8, 23, 5]. In the relational data model, a database instance I is viewed as a relational structure. Moreover, a query Q may involve additional predicates and functions on the atomic values that can appear in I. For example, consider the following SQL query over relations $R(A, B)$ and $S(C)$:

```
select A from R, S where B <= C
```

This query can be applied to instances that are logical structures involving, in addition to the database relations R and S, a less-than relation $\leq$.

The logical nature of queries then amounts to the general principle that a database query commutes with permutations of the atomic values that preserve all the relations over which the query is formulated. Such permutations are nothing else than isomorphisms of the relational structures. Thus, concisely, the principle can be stated as follows: *if Q is a query, I is an instance, and f is*

an isomorphism, then $Q(f(I)) = f(Q(I))$. Chandra and Harel [16] later called this principle "the consistency criterion" for database queries. The principle became finally known under the name *genericity* [20, 2]. Genericity can be easily adapted to data models other than the relational model, simply by adopting the appropriate notion of an isomorphism. Interestingly, genericity coincides with the definition Tarski proposed in 1966 of "logical notions" [25].

Commuting with isomorphisms is thus a property expected from all database queries, even the most complex ones. Simpler queries, however, may have stronger commutation properties. Well known, for example, is the class of queries that can be formulated without invoking the equality predicate; these queries can be characterized as those commuting not only with all isomorphisms, but more generally with all strong surjective homomorphisms [15].[1]

One the most stringent commutation property of queries one can consider was proposed by Beeri, Milo and Ta-Shma under the name of *full genericity* [9]. Recall that a query Q is generic in the classical sense if, for every instance I and permutation f of atomic values, we have $Q(f(I)) = f(Q(I))$. Now a query Q is called fully generic if the same holds for all functions f from atomic values to atomic values; so f does not need to be a permutation.

In order to get a feeling for full genericity, let us consider the operations of the relational algebra. Union, projection and cartesian product are fully generic, but selection, intersection and difference are not. More generally, one can develop the intuition that the fully generic database queries are those that combine or restructure the data without really having to look at the concrete data values. In particular, a fully generic query is not sensitive to the presence of duplicates in the input. Fully generic queries may produce a lot of output, but the amount of processing relative to the output size is typically minimal. For example, consider a Big Data setting, where data is distributed over different compute nodes. When performing a join, we need to ensure that joinable tuples reside on a common node [4]. For fully generic queries, however, no such requirement seems to be necessary.

A form of full genericity is also found in provenance queries, which track or propagate provenance annotations from the input to the output. Semantic characterizations of provenance queries [12] involve the property that annotations can be copied (or omitted), but are not to be compared with each other. Such a property is similar to full genericity, but applied only to the annotations.

In the relational data model, full genericity is a rather poor notion. Indeed, if we fix the input database schema and the output relation schema, there are only finitely many different fully generic queries. All we can do is form cartesian products of projections of database relations that produce results of the right output width, and take unions of these. When moving to the complex-object data model, however, the situation changes.

[1] Madelaine [22] has given a complete overview of the classes of morphisms corresponding to classes of queries expressible in different fragments of first-order logic, formed by the possible combinations of allowed features among existential quantification, universal quantification, conjunction, disjunction, negation, and equality.

The complex-object data model is a generalization of the relational data model. A relational database instance is essentially a tuple of relations, where each relation is a set of tuples of atomic values. The width of the database instance, i.e., the number of relations, and the widths of these relations, are given by the database schema. Note that the way tuple and set formations can be nested is completely fixed in the relational model: we have a tuple of sets of tuples of atomic values, nothing more, nothing less. Now moving to the complex-object model [2], we are allowed arbitrary combinations of tuple and set formation. Complex objects have been around since the early 1980s, and remain relevant for modern database systems. For example, the data model underlying Apache Spark [26, 7] is essentially that of complex objects, and also JSON databases such as MongoDB essentially store complex objects [11].

When thus considering queries over complex objects, where the types of inputs and outputs can be nested more and more deeply, the class of fully generic queries grows substantially. For a simple example, let k be a natural number. Given as input a set of atomic values, we may output the set of all subsets of the input having at most k elements. Each value of k gives rise to a different query, and all of these queries are fully generic. Note that all these queries have the same signature $\{d\} \to \{\{d\}\}$, i.e., they take as input a set of atomic values and they output a set of sets of atomic values (the symbol d stands for the atomic value type).

The goal of this paper is to draw attention to the fascinating class of fully generic queries. We find this class indeed fascinating because, while full genericity is a very stringent requirement, we still do not understand it well, especially when input and output types are deeply nested. What exactly are the fully generic queries? Some very basic questions about them remain unanswered:

1. Is every fully generic query effectively computable?
2. Can we effectively decide, given an input–output pair (A, B) of complex objects, whether there exists a fully generic query that maps A to B?
3. Is there a query language that captures the fully generic queries? (I.e., a language in which only fully generic queries can be expressed, but that is also complete, in that every computable fully generic query can be expressed.)

In contrast, the corresponding questions for classical generic queries have readily available answers: obviously negative for the first; affirmative for the second (just check if every automorphism of A is also an automorphism of B [8, 23]); and again affirmative for the third [16].

Question 3 originates from Beeri, Milo and Ta-Shma [9], who proposed as a candidate language the classical powerset algebra for complex objects [21, 1], from which we remove equality testing, and to which we add the intriguing operator *one-each*. Ta-Shma shows in her PhD thesis [24] that the resulting language, called $\mathcal{L}$, indeed captures the fully generic queries in two special but interesting cases: the case where the output is a flat relation, and the case where the signature is $\{\{d\}\} \to \{\{d\}\}$. Unfortunately, the given arguments are hard to verify. We will be able to offer a more transparent proof of the first result, and also of the special case $\{d\} \to \{\{d\}\}$. The language $\mathcal{L}$ also prompts various

further interesting questions, some already posed by Ta-Shma, which will be expounded below.

2 Complex Objects, Queries, and Genericity

In order to discuss the issues presented in the Introduction more formally, we start by defining the complex-object data model, the notion of a query, and the notions of classical and full genericity.

A *type* is an expression τ conforming to the following grammar:

$$\tau ::= d \mid [\tau, \ldots, \tau] \mid \{\tau\},$$

where d is a fixed symbol denoting the atomic value type. So, a type is either d, a tuple of types, or of the form $\{\tau\}$ with τ a type.

We assume, as given, a countably infinite domain **dom** of atomic values, or *atoms* for short. In examples, we will often use natural numbers for atoms. The set of *objects* is the smallest set such that

- every atom is an object;
- every tuple $[o_1, \ldots, o_k]$ of objects is an object; and
- every finite set $\{o_1, \ldots, o_n\}$ of objects is an object.

We will work only with well-typed objects. Informally, these are objects where in every set, all its elements are of the same kind. For example, the set $\{1, [2], [1, 2], \{1, 2, 3\}\}$ is very badly typed: its four elements are all of different kinds (an atom, a one-tuple, a two-tuple, and a set of atoms, respectively). Formally, an object o is said to be *of type* τ if one of the following holds:

- τ is d and o is an atom;
- τ is a tuple type $[\tau_1, \ldots, \tau_k]$ and o is a k-tuple $[o_1, \ldots, o_k]$ with o_i of type τ_i for $i = 1, \ldots, k$;
- τ is a set type $\{\tau'\}$ and o is a finite set of objects of type τ'.

We will denote the set of objects of type τ by $[\![\tau]\!]$.

For types τ_{in} and τ_{out}, we now define a *query* of signature $\tau_{\text{in}} \to \tau_{\text{out}}$, quite simply, to be a total function q from $[\![\tau_{\text{in}}]\!]$ to $[\![\tau_{\text{out}}]\!]$. We will denote this by $q : \tau_{\text{in}} \to \tau_{\text{out}}$. We say that q is

- *generic* if we have $q(f(D)) = f(q(D))$, for every object D of type τ_{in} and every permutation f of **dom**.
- *fully generic* if we have $q(f(D)) = f(q(D))$, for every object D of type τ_{in} and every function $f : \mathbf{dom} \to \mathbf{dom}$.

Here, by $f(D)$, we naturally mean the object obtained from D by replacing each atom x by the atom $f(x)$.

The notion of genericity is classical and, for extensive discussion and motivation, we refer to the literature cited in the Introduction, and also to the work by Abiteboul and Vianu on generic computation [3] and by Blass, Gurevich

and Shelah on Choiceless Polynomial Time [10], on which quite a bit of recent follow-up work has been performed [18].

In order to get a feeling for full genericity, let us look at two simple examples. First, consider the query $q : \{[d, d]\} \to \{[d, d, d]\}$ that takes as an input a binary relation of atoms. It outputs the ternary relation obtained from the input by swapping the two columns, and duplicating the second column in a third column. So, formally, $q(D) = \{[y, x, y] \mid [x, y] \in D\}$. We may view q as the projection operator $\pi_{2,1,2}$ from relational algebra. This query is readily verified to be fully generic:

$$\begin{aligned} q(f(D)) &= \{[y, x, y] \mid [x, y] \in f(D)\} \\ &= \{[f(v), f(u), f(v)] \mid [u, v] \in D\} \\ &= f(q(D)). \end{aligned}$$

In contrast, the query $q : [\{d\}, \{d\}] \to \{d\}$ that takes as an input two sets of atoms, and outputs their intersection, is not fully generic. Indeed, just consider the input $D = [\{1\}, \{2\}]$. Then $q(D)$ is empty, so also $f(q(D))$ is empty for any f. However, for f that maps both 1 and 2 to 1, we obtain $q(f(D)) = q([\{1\}, \{1\}]) = \{1\}$, which is nonempty.

3 Computability

The standard notion of computability, through Turing machines, is only defined as such for functions from Σ^* to Σ^*, for some finite alphabet Σ. Consequently, computability of queries needs a proper definition [2], which we recall next.

We first need to fix some encoding of atoms as strings; as usual, binary strings will suffice. So, assume some bijection $enc : \mathbf{dom} \to \{0, 1\}^*$. We can now consider the finite alphabet obtained by extending $\{0, 1\}$ with the punctuation symbols needed to write down objects: the comma, the square brackets, and the curly brackets. So, $\Sigma = \{0, 1\} \cup \{\texttt{,}, \texttt{[}, \texttt{]}, \texttt{\{}, \texttt{\}}\}$. We can similarly consider the *infinite* alphabet Λ obtained by adding these punctuation symbols to **dom** directly. For any object o, an *enumeration* of o is a string over Λ that describes o when interpreted in the obvious manner. For example,

- The only enumeration of the object $[9, 7]$ is the string `[9,7]`.
- The three strings `{1,2,3}`, `{3,2,1}`, and `{2,1,2,1,3,3}` all enumerate the same object $\{1, 2, 3\}$.

When we apply *enc* to an enumeration of an object o, we obtain what we call an *encoding* of o. For example, we can take again the first example above, and assume *enc* is the standard binary representation of natural numbers. Then the string `[1001,111]` encodes the object $[9, 7]$.

We now agree that a query $q : \tau_{\text{in}} \to \tau_{\text{out}}$ is *computable under enc* if there exists a Turing machine M that, whenever started on an input that is an encoding of some object o of type τ_{in}, will eventually halt and produce an encoding of $q(o)$ as output. In this case we also say that M *computes* q *under enc*.

The nice thing about generic queries is that the choice of encoding does not matter:

Proposition 1. *Let q be a generic query and let enc : $\mathbf{dom} \to \{0,1\}^*$ be a bijection. Let M be a Turing machine. If M computes q under enc, then M also computes q under any other bijection enc′ : $\mathbf{dom} \to \{0,1\}^*$.*

In line with the above proposition, Hull and Su [19] proposed the notion of a *domain Turing machine*, which can work directly over the alphabet Λ. Thereto, the machine is equipped with a register that can hold an arbitrary atom. The machine can copy the atom from the current tape cell into the register, and conversely can copy the atom from the register into the current tape cell. Furthermore, the machine can test for equality between the atom in the register and the atom in the current tape call. Since domain Turing machines can directly take enumerations of objects as inputs, and can produce such enumerations as outputs, we no longer need encodings. Now Hull and Su showed that if a domain Turing machine computes a query q, then q must be generic, and conversely, every computable generic query can be computed by some domain Turing machine.

For fully generic queries, we can strengthen the Hull-Su result as follows. A *domain-oblivious* Turing machine is a restricted domain Turing machine that lacks the facility to test for equality in the sense described in the previous paragraph.

Theorem 1. *If query q is computed by a domain-oblivious Turing machine, then q is fully generic. Conversely, every fully generic computable query can be computed by some domain-oblivious Turing machine.*

The above result provides some insight into the notion of a fully generic query. It confirms the intuition that to process a fully generic query, we never need to inspect atoms in detail. We merely copy them or omit them altogether. Good examples are the relational algebra operations union, projection, and cartesian product. Furthermore, a fully generic query should not be sensitive to duplicates in the input, as these are allowed in enumerations as defined above.

Example 1 (Duplicates). To illustrate the sensitivity to duplicates, consider the query $q_1 : \{\{d\}\} \to \{\{d\}\}$ defined as follows. Let D be an input object, $D = \{s_1, \ldots, s_n\}$, where all the sets s_i are distinct. Then $q_1(D)$ consists of all the sets that can be written as $\{o_1, \ldots, o_n\}$ such that $o_i \in s_i$ for $i = 1, \ldots, n$. Note that the o_is picked need not be all distinct; for example, assuming $D = \{\{1,2\},\{1,3\}\}$, we can pick $o_1 = 1 \in \{1,2\}$ and $o_2 = 1 \in \{1,3\}$. Thus, the set $\{1,1\} = \{1\}$ belongs to $q_1(\{\{1,2\},\{1,3\}\})$.

This query is not fully generic, intuitively, because the given prescription for computing q_1 assumes that all the s_i are distinct. For example, suppose $D = \{\{1,2\}\}$. Then $q_1(D) = \{\{1\},\{2\}\}$. However, if we had presented D as an input with duplicates, say $\{\{1,2\},\{1,2\}\}$, the above prescription could generate $\{1,2\}$ as a possible element of the result, which is wrong.

We can formalize the above observation as follows. Let $D' = \{\{1,2\},\{3,4\}\}$ and take some $f : \mathbf{dom} \to \mathbf{dom}$ such that $f(1) = 1$, $f(2) = 2$, $f(3) = 1$, and $f(4) = 2$. Since $\{1,4\} \in q_1(D')$, we have $f(\{1,4\}) = \{1,2\} \in f(q_1(D'))$. However, $f(D') = \{\{1,2\}\}$ and $\{1,2\} \notin q_1(f(D'))$. Hence, q_1 is not fully generic.

Theorem 1 begs the following question, which is embarrassingly open:

Question 1. Do there exist fully generic queries that are not computable?

The conjecture is that the answer is negative. We even dare to conjecture that every fully generic query is computable in time linear in the output size. (Here, to get a useful notion of linear time, we would need to move from a Turing machine model to a RAM model of computation.) This conjecture is in line with the intuition that the processing is done in a manner that is oblivious to the actual identities of the atoms. Thus, all the processing time can be devoted to producing the output.

Boolean Queries and Canonical Forms Restricted to Boolean queries, the above question has quite readily a negatie answer. A Boolean query has just a yes/no answer and can be modeled as a query $q : \tau \to \{[\,]\}$. Indeed, there are only two objects of type $\{[\,]\}$: the empty set and the singleton set containing the empty tuple. The empty set can be taken to represent 'no' and the other set 'yes'.

To see that any fully generic $q : \tau \to \{[\,]\}$ is computable, let D be an object of type τ and let $\mathbf{1}$ be the mapping that maps every atom to 1. We have $q(\mathbf{1}(D)) = \mathbf{1}(q(D)) = q(D)$. Consequently, we fully know the behavior of q once we know the behavior of q on objects in which only the atom 1 occurs. Such objects are called *canonical forms* [24]. For any given type τ, there are only finitely many canonical forms of type τ. Thus, every fully generic Boolean query can be summarized in a finite table and hence is always computable.

As an example, consider the type $\{\{d\}\}$. There are only four canonical forms: $\emptyset$; $\{\emptyset\}$; $\{\{1\}\}$; and $\{\emptyset,\{1\}\}$. For example, the canonical form of the object $\{\{1,2\},\{2,3,4\},\emptyset\}$ is $\{\emptyset,\{1\}\}$; the canonical form of $\{\{1,2\},\{2,3,4\},\{5\}\}$ is $\{\{1\}\}$. As a consequence, there are exactly $2^4 = 16$ fully generic Boolean queries with input type $\{\{d\}\}$.

We can characterize when two objects have the same canonical form in terms of a pre-order $A \leq B$ on objects of the same type. We define $A \leq B$ to hold when there exists a function $f : \mathbf{dom} \to \mathbf{dom}$ such that $A = f(B)$. We can now show the following.

Proposition 2. *Let A and B be objects of the same type. Then $\mathbf{1}(A) = \mathbf{1}(B)$ if and only if A and B have a common upper bound w.r.t. $\leq$, i.e., if there exists an object C such that $A \leq C$ and $B \leq C$.*

The 'if' implication is immediate; if $A = f(C)$ then $\mathbf{1}(A) = \mathbf{1}(f(C)) = \mathbf{1}(C)$, and similarly for B. The 'only if' implication is less trivial.[2]

[2] A comparable result was shown by Ta-Shma [9, Claim 3.4], [24, Proposition 4.2.4].

The Definability Question Another computability question concerns definability by a fully generic query. This is the following problem:

Problem: Fully generic definability
Input: Two objects A and B
Decide: Does there exist a fully generic query q such that $q(A) = B$?

Question 2. Is the fully generic definability problem decidable?

In contrast, the corresponding classically generic definability problem is well understood. For simplicity, assume B is of some set type. Then A and B qualify if and only if B has only atoms from A, and every automorphism of A is also an automorphism of B.[3] In that case, the generic query mapping A to B can even taken to be expressible in first-order logic [8, 23, 6].

4 Query Language

More concrete insight in the fully generic queries can be gained by studying the language $\mathcal{L}$ already mentioned in the Introduction [9]. This language is an algebra of queries, similar to the powerset algebra of Abiteboul and Beeri [1], presented in monad style [13]. The algebra is built up from the following list of primitive queries, for all types τ, σ, τ_1, ..., τ_k:

- The identity query $id : \tau \to \tau : o \mapsto o$.
- The unit query $[\,] : \tau \to [\,] : o \mapsto [\,]$ which always outputs the empty tuple on every input.
- For each $i \in \{1, \ldots, k\}$, the projection $\pi_i : [\tau_1, \ldots, \tau_k] \to \tau_i : [o_1, \ldots, o_k] \mapsto o_i$.
- The empty-set query $\emptyset : \tau \to \{\sigma\} : o \mapsto \emptyset$, which always outputs the empty set of type σ.
- The singleton query $\{\cdot\} : \tau \to \{\tau\} : o \mapsto \{o\}$.
- The flatten query $\bigcup : \{\{\tau\}\} \to \{\tau\} : o \mapsto \bigcup o$.
- The union query $\cup : [\{\tau\}, \{\tau\}] \to \{\tau\} : [o_1, o_2] \mapsto o_1 \cup o_2$.
- The cartesian product $\times : [\{\tau\}, \{\sigma\}] \to \{[\tau, \sigma]\} : [o_1, o_2] \mapsto o_1 \times o_2$.
- The emptiness test

$$\mathit{ifempty} : [\{\sigma\}, \tau, \tau] \to \tau : [s, o_1, o_2] \mapsto \begin{cases} o_1 & \text{if } s \text{ is empty} \\ o_2 & \text{if } s \text{ is not empty.} \end{cases}$$

Moreover, we close the algebra under composition, tuple construction, and *map*, as follows:

- If $q_1 : \tau_1 \to \tau_2$ and $q_2 : \tau_2 \to \tau_3$ belong to $\mathcal{L}$, then so does the composition $q_2 \circ q_1 : \tau_1 \to \tau_3$.

[3] Here, an automorphism of A is a permutation f of the atoms occurring in A such that $f(A) = A$.

– If $q : \tau \to \sigma$ belongs to $\mathcal{L}$, then so does

$$map(q) : \{\tau\} \to \{\sigma\} : o \mapsto \{q(o') \mid o' \in o\}.$$

– If, for $i = 1, \ldots, k$, we have $q_i : \sigma \to \tau_i$ in $\mathcal{L}$, then also $[q_1, \ldots, q_k] : \sigma \to [\tau_1, \ldots, \tau_k] : o \mapsto [q_1(o), \ldots, q_k(o)]$ belongs to $\mathcal{L}$.

So far, we have done nothing else than described the standard nested relational algebra [13], without equality test. However, the definition of the language $\mathcal{L}$ must be completed by adding to the above list of primitive queries, for every type τ, the query *one-each* : $\{\{\tau\}\} \to \{\{\tau\}\}$ defined by

$$\{s_1, \ldots, s_n\} \mapsto \{s'_1 \cup \cdots \cup s'_n \mid \emptyset \neq s'_i \subseteq s_i \text{ for } i = 1, \ldots, n\}.$$

The query *one-each* is quite intriguing and, compared to the powerset algebra mentioned above, the only novel aspect of the language $\mathcal{L}$. Actually, a more correct name would be *at-least-one-each*, but we stick to the original name. The query *one-each* is certainly primitive in $\mathcal{L}$, as it is the only query from $\mathcal{L}$ that can produce exponential-sized outputs. Indeed, when applied to just a singleton $\{s\}$, the output is already the powerset of s, except for the empty set. Conversely, however, it is conjectured that *one-each* can not be replaced by the powerset query, but a proof is lacking so far:

Question 3. Let $\mathcal{L}^{pow}$ denote the variant of $\mathcal{L}$ where we replace *one-each* by the powerset query $pow : \{\tau\} \to \{\{\tau\}\} : s \mapsto 2^s$. Does *one-each* belong to $\mathcal{L}^{pow}$?

Of course, the million-dollar question is the following:

Question 4. Does $\mathcal{L}$ contain all the fully generic queries?

An affirmative answer to this question would immediately yield a negative answer to Question 1 on the existence of noncomputable fully generic queries.

The only investigation on Question 4 so far was done in the PhD thesis by Paula Ta-Shma [24]. Her main result is that $\mathcal{L}$ does contain all the fully generic queries of signature $\{\{d\}\} \to \{\{d\}\}$. While her arguments are rich in valuable ideas, the arguments are also very intricate, and at some point no longer fully rigorous. We have failed to verify the arguments in detail; nevertheless, the thesis is a must-read for anyone interested in solving the above open question.

In our attempt to find more transparent arguments, we could prove the following result. For any natural number k, define the *k-powerset* of a set s as the set of all subsets of s of cardinality at most k.

Theorem 2. *The only fully generic queries of signature* $\{d\} \to \{\{d\}\}$ *are:*

– *the powerset query;*
– *for any* k*, the* k*-powerset query;*
– *for any of the above queries* q*, also the queries* $q^{(0)} : s \mapsto q(s) - \{\emptyset\}$*;* $q^{(1)} : s \mapsto q(s) \cup \{s\}$*; and* $q^{(2)} : s \mapsto (q(s) - \{\emptyset\}) \cup \{s\}$*.*

Note that all queries mentioned in the above theorem belong to $\mathcal{L}$, so this answers Question 4 for the special case of the signature $\{d\} \to \{\{d\}\}$.

As already mentioned, the language $\mathcal{L}$ without *one-each* is the standard nested relational algebra without equality test. Equivalence of nested relational algebra expressions is well known to be undecidable. When emptiness test is removed, and equality test is restricted to atoms, equivalence becomes decidable [14]. However, the equivalence problem when keeping emptiness test, but removing equality test altogether, seems to have escaped attention so far. We have the following interesting questions:

Question 5. Is the equivalence problem for expressions in $\mathcal{L}$ without *one-each* decidable? What about $\mathcal{L}$ including *one-each*? And what about $\mathcal{L}^{pow}$?

5 Technical Observations

A possible approach to comprehending the fully generic queries better is to investigate how large inputs we must consider to show the difference between two fully generic queries. For example, we cannot see the difference between the 5-powerset and the 6-powerset by considering only input sets with at most five elements. We can also investigate how many different atoms must come into play.

Consider, for example, the behavior of *one-each* : $\{\{d\}\} \to \{\{d\}\}$ on inputs in which at most two distinct atoms can occur. We can show:

Proposition 3. *Let $q' : \{\{d\}\} \to \{\{d\}\}$ be fully generic, such that $q'(D) =$ one-each(D) for every D in which at most two distinct atoms appear. Then q' equals one-each.*

We summarize the above result by saying that *one-each* is *2-determined*. In general, we say that a query q is *k-determined* if no other fully generic query agrees with q on all inputs involving at most k distinct atoms. If q is k-determined for some k, we also say that q is *finitely determined*.

For example, the k-powerset query of signature $\{d\} \to \{\{d\}\}$ is $(k+1)$-determined. In contrast, the powerset query is not finitely determined, because for any k, it agrees up to k atoms with the k-powerset query.

A special class of queries, also considered by Ta-Shma, are the *flat-output* queries, defined as those of signature of the form $\tau \to \{[d, \ldots, d]\}$ (the output is a flat relation). We can show:

Theorem 3. *Let q be a fully generic flat-output query and let k be the width of its output tuple type. Then q is $(k+1)$-determined.*

As a corollary, we obtain a more transparent proof of the result by Ta-Shma to the effect that every fully generic flat-output query belongs to $\mathcal{L}$. We only sketch the argument in this conference paper. Fix some ℓ atoms and, for a signature $\tau_{\text{in}} \to \tau_{\text{out}}$, define $[\![\tau_{\text{in}}]\!]^{(\ell)}$ and $[\![\tau_{\text{out}}]\!]^{(\ell)}$ as the (finite) subsets of $[\![\tau_{\text{in}}]\!]$ and $[\![\tau_{\text{out}}]\!]$ consisting of the objects involving only the ℓ given atoms. There are only finitely

many fully generic functions $q : [\![\tau_{\text{in}}]\!]^{(\ell)} \to [\![\tau_{\text{out}}]\!]^{(\ell)}$, and these can be represented in $\mathcal{L}$.

In general, the usefulness of finite determinacy remains unclear, as it is not preserved by composition. For example, both $\{\cdot\}$ and *one-each* are finitely determined, but their composition, the powerset query, is not.

6 Conclusion

This paper is an invitation, a "call to arms", for a renewed investigation of the forgotten, but very natural and fascinating class of fully generic queries. Indeed, we have characterized these queries as those that can be processed in a domain-oblivious manner, and are insensitive to duplicates in the input. Various data transformations or restructurings have this property. We have posed the main open questions and have proposed some possible approaches. We are looking forward to answers appearing in the near future.

It should also be investigated how full genericity behaves in a setting where collections are bags instead of sets. For example, the bag version of the query q_1 from Example 1, where we do not assume that all the s_i are distinct, is fully generic in the bag setting. Some of the questions we have posed may become easier in the bag setting, but others (e.g., Question 5) may become more difficult.

References

1. Abiteboul, S., Beeri, C.: On the power of languages for the manipulation of complex objects. The VLDB Journal **4**(4), 727–794 (1995)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
3. Abiteboul, S., Vianu, V.: Computing with first-order logic. J. Comput. Syst. Sci. **50**(2), 309–335 (1995)
4. Afrati, F., Ullman, J.: Optimizing multiway joins in a map-reduce environment. IEEE Transactions on Knowledge and Data Engineering **23**(9), 1282–1298 (2011)
5. Aho, A., Ullman, J.: Universality of data retrieval languages. In: Conference Record, 6th ACM Symposium on Principles of Programming Languages. pp. 110–120 (1979)
6. Arenas, M., Diaz, G.: The exact complexity of the first-order logic definability problem. ACM Trans. Database Syst. **41**(2), 13:1–13:14 (2016)
7. Armbrust, M., Xin, R., et al.: Spark SQL: Relational data processing in Spark. In: Proceedings 2015 International Conference on Management of Data. pp. 1383–1394. ACM (2015)
8. Bancilhon, F.: On the completeness of query languages for relational data bases. In: Proceedings 7th Symposium on Mathematical Foundations of Computer Science. Lecture Notes in Computer Science, vol. 64, pp. 112–123. Springer-Verlag (1978)
9. Beeri, C., Milo, T., Ta-Shma, P.: Towards a language for the fully generic queries. In: Cluet, S., Hull, R. (eds.) Database Programming Languages. pp. 239–259. Lecture Notes in Computer Science, Springer (1997)
10. Blass, A., Gurevich, Y., Shelah, S.: Choiceless polynomial time. Annals of Pure and Applied Logic **100**, 141–187 (1999)

11. Botoeva, E., Calvanese, D., Cogres, B., Xiao, G.: Expressivity and complexity of MongoDB queries. In: Kimelfeld, B., Amsterdamer, Y. (eds.) Proceedings 21st International Conference on Database Theory. LIPIcs, vol. 98, pp. 9:1–9:23. Schloss Dagstuhl–Leibniz Center for Informatics (2018)
12. Buneman, P., Cheney, J., Vansummeren, S.: On the expressiveness of implicit provenance in query and update languages. ACM Trans. Database Syst. **33**(4), 28:1–28:47 (2008)
13. Buneman, P., Naqvi, S., Tannen, V., Wong, L.: Principles of programming with complex objects and collection types. Theor. Comput. Sci. **149**(1), 3–48 (1995)
14. Van den Bussche, J., Van Gucht, D., Vansummeren, S.: Well-definedness and semantic type checking for the nested relational calculus. Theor. Comput. Sci. **371**(3), 183–199 (2007)
15. Chandra, A.: Programming primitives for database languages. In: Conference Record, 8th ACM Symposium on Principles of Programming Languages. pp. 50–62 (1981)
16. Chandra, A., Harel, D.: Computable queries for relational data bases. J. Comput. Syst. Sci. **21**(2), 156–178 (1980)
17. Garcia-Molina, H., Ullman, J., Widom, J.: Database Systems: The Complete Book. Prentice Hall (2009)
18. Grädel, E., Grohe, M.: Is polynomial time choiceless? In: Beklemishev, L., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II, pp. 193–209. Springer (2015)
19. Hull, R., Su, J.: Algebraic and calculus query languages for recursively typed complex objects. J. Comput. Syst. Sci. **47**(1), 121–156 (1993)
20. Hull, R., Yap, C.: The format model, a theory of database organization. J. ACM **31**(3), 518–537 (1984)
21. Kuper, G., Vardi, M.: The logical data model. ACM Trans. Database Syst. **18**(3), 379–413 (1993)
22. Madelaine, F.: Mémoire d'habilitation à diriger des recherches, Université Blaise Pascal, Clermond-Ferrand (2012), https://tel.archives-ouvertes.fr/tel-01096078
23. Paredaens, J.: On the expressive power of the relational algebra. Inf. Process. Lett. **7**(2), 107–111 (1978)
24. Ta-Shma, P.: Genericity in Database Query Languages. Ph.D. thesis, Hebrew University (1997)
25. Tarski, A.: What are logical notions? History and Philosophy of Logic **7**, 143–154 (1986), edited by J. Corcoran
26. Zaharia, M., et al.: Spark: Cluster computing with working sets. In: Proceedings 2nd USENIX Workshop on Hot Topics in Cloud Computing (2010)