Analyzing the Behavior of Database Applications Through Size Descriptions

Jan Van den Bussche[1] and Emmanuel Waller[2(✉)]

[1] Data Science Institute, Hasselt University, Hasselt, Belgium
jan.vandenbussche@uhasselt.be
[2] LISN, Paris-Saclay University, Gif-sur-Yvette, France
waller@lisn.fr

Abstract. We propose a description language capturing a simple but central aspect of the possible "lives" (the behavior) of a database application, namely the ability for the relations to grow, shrink, or keep the same size. We first consider arbitrary behaviors and investigate the basic properties of such descriptions. We characterize consistency, redundancy, and subsumption, and show that every behavior has a unique minimal complete description. We then consider the problem of computing the minimal complete description of a given database application. We model such applications as collections of procedures, specified by update programs based on relational algebra. The general problem is undecidable, even for an application consisting of a single procedure with a single update language statement. We also identify decidable cases and provide partial complexity characterisations.

Keywords: Dynamics of information · Static analysis of update programs

1 Introduction

The analysis of database evolution is an important area of research [6]. Since the database reflects the relevant processes that take place in a business or enterprise, a good description of how the database evolves provides valuable information to the business analyst. Standard approaches use various forms of temporal logic to express properties of database evolutions, e.g., [1,7,8,11]. In the present paper, we explore an alternative approach, which is more coarse-grained and is focused on the size changes of the tables in the database.

Indeed, our hypothesis is that information about which tables grow, shrink, or keep the same size already provides useful information on the behavior of various business processes. An information system is typically organized as a collection of related entities; data about the entities and their relationships is stored in various tables [10]. Each entity has a lifecycle. When the entity is created, this is reflected in the growth of one or more entity tables. Relationships between entities will be created, which is again reflected in the growth of the

A.-Y. Turhan and J. Virtema (Eds.): FoIKS 2026, LNCS 16475, pp. 195–210, 2026.
https://doi.org/10.1007/978-3-032-21540-6_12

corresponding tables. Similarly, relationships that end cause certain tables to shrink. For example, on booking.com, customers and properties are entities, and customers booking certain properties are relationships. When detailed attributes of entities or relationships are updated, this typically only modifies the entries of a table row and will reflect as the table keeping its size. Finally, entities may be withdrawn, again reflected as a shrink in a table.

In this paper, we begin by developing a general theory of size descriptions. We consider arbitrary evolutions of database states, which we call database lives. We introduce a description language where formulas express how the sizes of tables change in the transitions between successive states in a life. We investigate the basic properties of descriptions, providing characterizations of consistency, redundancy, and subsumption. We introduce the general notion of *behavior* as a (possibly infinite) set of lives. In our framework, every behavior will turn out to have a unique minimal complete description.

In the second part of this paper, we investigate the problem of automatically deriving the minimal complete description of the behavior of database applications. We introduce a formal model of real-life database applications, where database queries and updates are embedded in imperative code. We formalize queries using relational algebra, and model SQL updates (insert, delete, update) using the formalism of update programs already proposed in our earlier work with Ameloot [4].[1] The novel feature of our model is that it has explicit notions of *procedures*, written in the language of update programs, and *procedure calls* with parameters. A similar model of embedded SQL was proposed by Itzhaky et al. [9], not with our focus on analyzing the lives of an application, but rather for the purpose of verifying individual programs by inferring weakest preconditions.

Since our model allows arbitrary relational algebra expressions as queries, it is not unexpected that the general problem of inferring the minimal complete description of a given application is undecidable. Still, our description language is novel, and we aim to delinate the boundary of undecidability. Thereto we need to develop original proof techniques that establish connections between various forms of satisfiability, various update operations, and various size-change primitives. Moreover, we will also identify cases where the problem is decidable.

To our knowledge, this work is the first to analyze database evolutions through the lens of size changes. Hence, this paper opens many directions for further research. These are discussed in the conclusion.

This paper is organized as follows. Section 2 establishes some basic terminology and notation. Section 3 introduces our description language and develops the general theory. Section 4 introduces the formal model of database applications. Section 5 presents decidability and undecidability results on inferring minimal complete descriptions of applications. Section 6 concludes. Proofs are omitted due to space limitations.

[1] Seminal work on update and transaction languages was done by Abiteboul and Vianu [3].

2 Preliminaries

We recall some basic notions and terminology from the theory of relational databases [2]. From the outset we assume a countably infinite universe **dom** of data elements. For a natural number k, a k-ary *relation* is a finite subset of $\mathbf{dom} \times \cdots \times \mathbf{dom}$ (k times), or, in other words, a finite set of k-tuples of data elements.

A *database schema* is a nonempty finite set **S** of *relation names* where every relation name has an associated arity (a natural number). An *instance* I of **S** is an assignment of a relation $I(R)$ to every relation name $R \in \mathbf{S}$, so that if R has arity k then $I(R)$ is k-ary. The set of all elements from **dom** occurring in I is called the *active domain* of I and denoted by $adom(I)$.

Remark 1. In the general framework, arities will not play a role, but they will play a role in our investigation of database applications.

3 General Theory of Size Descriptions

3.1 Database Lives

Let **S** be a schema. A *database life* over **S** (or *life* for short) is a finite sequence of at least two instances of **S**, starting with the empty instance (i.e., the instance where all relations are empty).

An ordered pair (I, I') of instances (of the same schema), also written simply as I, I', is called a *transition*. If I and I' occur consecutively in a life l, we say that the transition *occurs in* l.

Remark 2. One may wonder why we insist that a life starts with the empty instance. Our goal is to model database evolutions which are typically governed by an application. (We will formalize applications later in this paper.) Every such application will start with a "filling" stage providing an initial operational state of the database. Our assumption that a life starts with the empty instance is equivalent to assuming that the filling process is part of the application. Even when the initial state is imported from somewhere else, it is reasonable to assume that it is semantically coherent, i.e., it could have been generated by the filling process.

3.2 Descriptions

Let **S** be a schema. A *size primitive* over **S** (or *primitive* for short) is a term of the form $R^{\nearrow}$, $R^{\searrow}$, or $R^{=}$, with R a relation name in **S**. These primitives are referred to as *grow*, *shrink*, and *same-size*, respectively. A transition I, I' *satisfies* a primitive p, denoted by $I, I' \models p$, in the following cases:

- $p = R^{\nearrow}$ and $|I(R)| < |I'(R)|$;
- $p = R^{\searrow}$ and $|I(R)| > |I'(R)|$;
- $p = R^{=}$ and $|I(R)| = |I'(R)|$.

Here, given any set X, we denote by $|X|$ the *size* of X, i.e., the number of its elements.

A *formula* over $\mathbf{S}$ is a set of primitives over $\mathbf{S}$ containing, for every relation name R in $\mathbf{S}$, exactly one primitive.

Example 1. Over a schema consisting of two relation names R and S, all possible formulas are $\{R^{\nearrow}, S^{\nearrow}\}$; $\{R^{\nearrow}, S^{\searrow}\}$; $\{R^{\nearrow}, S^{=}\}$; $\{R^{\searrow}, S^{\nearrow}\}$; $\{R^{\searrow}, S^{\searrow}\}$; $\{R^{\searrow}, S^{=}\}$; $\{R^{=}, S^{\nearrow}\}$; $\{R^{=}, S^{\searrow}\}$; and $\{R^{=}, S^{=}\}$. □

A transition I, I' *satisfies* a formula F, denoted by $I, I' \models F$, if I, I' satisfies all primitives in F.

Two remarks are in order:

Remark 3. 1. From the above definition, we see that a formula is interpreted as a *conjunction* of primitives.
2. Clearly, every transition satisfies exactly one formula.

Definition 1. *A* description *over* $\mathbf{S}$ *is a set of formulas over* $\mathbf{S}$.

A life l satisfies a description D, denoted by $l \models D$, if every transition in l satisfies some formula in D.

Remark 4. By the above definition, a description is interpreted as a *disjunction* of formulas. For clarity, in examples we will often write descriptions as disjunctions. For example, over a single relation name R, we may write the "tautological" description $\{\{R^{\nearrow}\}, \{R^{\searrow}\}, \{R^{=}\}\}$ as $\{R^{\nearrow}\} \vee \{R^{\searrow}\} \vee \{R^{=}\}$. We can then also write $\{\{R^{=}\}\}$ simply as $\{R^{=}\}$.

Definition 2. *The set of all lives satisfying a given description D is denoted by $lives(D)$.*

3.3 Fundamental Properties of Descriptions

In the following we assume some fixed arbitrary schema $\mathbf{S}$ is given.

Consistency

Definition 3. *A description D is called* consistent *if $lives(D)$ is not empty.*

Example 2. A simple example of an inconsistent description, over a single relation name R, has only the formula $\{R^{\searrow}\}$. Indeed, any life starts with the empty instance, and no transition can start from the empty instance and satisfy $R^{\searrow}$. A simple example of a consistent description is $\{R^{\nearrow}\}$. The tautological description $\{R^{\nearrow}\} \vee \{R^{\searrow}\} \vee \{R^{=}\}$ is also consistent (every possible life satisfies it). Finally, note that $\{R^{=}\}$ is also consistent, but the only lives that satisfy it are sequences of the empty instance. □

The above example suggests the following characterization. We say that a formula is *grow only* if it does not contain any shrink primitive.

Proposition 1. *A description is consistent if and only if it contains a grow-only formula.*

Realizable Formulas and Reduced Descriptions. Let us say that a formula F is *realized* in a life l if some transition in l satisfies F. Now a formula F in a description D is said to be *realizable with respect to* D if there exists a life that satisfies D and that realizes F. Intuitively, a formula in D that is unrealizable with respect to D serves no purpose in D and can be removed from D.

Example 3. Over two relation names R and S, consider the description $D = F_1 \vee F_2$ with $F_1 = \{R^{\nearrow}, S^{=}\}$ and $F_2 = \{R^{=}, S^{\searrow}\}$. Clearly, F_1 is realizable with respect to D as shown by any life I_0, I_1 where I_0 is empty, R is nonempty in I_1, and S is empty in I_1. In contrast, F_2 is unrealizable with respect to D. Indeed, since D contains no formula where S grows, and every life starts with the empty instance, S will be empty in every instance of every life that satisfies D. Hence, no transition in such a life can shrink S. □

The above example suggests that a formula F is unrealizable with respect to D if F contains a shrink primitive for a relation that cannot be made nonempty. We formalize this by the following syntactic notion of "fillable". The link with realizability will be established by Theorem 1.

Definition 4. *Let D be a description. The set $\mathcal{F}$ of* fillable *relation names with respect to D is the smallest set of relation names from* **S** *with the following property. If there exists a formula $F \in D$ such that $R^{\nearrow} \in F$ and for every $S^{\searrow} \in F$ we have $S \in \mathcal{F}$, then $R \in \mathcal{F}$.*

Remark 5. The set of fillable relation names can be constructed inductively. We initialize $\mathcal{F}_0 = \emptyset$. For any natural number k, we define $\mathcal{F}_{k+1}$ as the set of relation names that occur growing in some formula that only contains shrink primitives for relations in $\mathcal{F}_k$. For example, to build $\mathcal{F}_1$, we can only use grow-only formulas. We obtain $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \cdots \subseteq \mathcal{F}_k$ until some k such that $\mathcal{F}_k = \mathcal{F}_{k+1}$, which exists since the number of relation names is finite. This $\mathcal{F}_k$ equals the set of fillable relation names.

Example 4. Consider the description

$$\{R_1^{\nearrow}, R_2^{\nearrow}, R_3^{\searrow}, R_4^{\nearrow}\} \vee \{R_1^{\searrow}, R_2^{\nearrow}, R_3^{=}, R_4^{=}\} \vee \{R_1^{\nearrow}, R_2^{=}, R_3^{=}, R_4^{=}\}.$$

The fillable relations are R_1 and R_2. □

We arrive at the following characterization:

Theorem 1. *F is realizable with respect to D if and only if D is consistent and for every $S^{\searrow}$ in F we have that S is fillable with respect to D.*

The theorem can be proven from the two following lemmas. (Lemma 1 for the only-if direction, and Lemma 2 for the if-direction.)

Lemma 1. *Assume R is nonempty in an instance in a life that satisfies some description D. Then R is fillable with respect to D.*

Lemma 2. *Let D be a consistent description. For every natural number n there exists a life satisfying D and containing an instance where all relations that are fillable with respect to D have size at least n.*

The above Lemma is stated stronger than needed for the if-direction of Theorem 1, which needs only a life satisfying D and ending in an instance where all fillable relations are nonempty. The stronger form is needed to allow an inductive proof that follows the inductive construction from Remark 5 and that correctly deals with formulas involving shrinks. Moreover, the stronger form will also be needed in the proof of Lemma 3.

We say that a description D is *reduced* if all formulas in D are realizable with respect to D. Theorem 1 implies the following characterization:

Proposition 2. *D is reduced if and only if every relation name that is mentioned as growing or shrinking in D is fillable.*

The reduced descriptions are the ones that are "minimal" in the following sense:

Proposition 3. *D is reduced if and only if $\mathit{lives}(D') \subsetneq \mathit{lives}(D)$ for every $D' \subsetneq D$.*

Subsumption. For descriptions D_1 and D_2 it is natural to say that D_2 *subsumes* D_1, denoted by $D_1 \preceq D_2$, if $\mathit{lives}(D_1) \subseteq \mathit{lives}(D_2)$. Clearly, if $D_1 \subseteq D_2$ then $D_1 \preceq D_2$. The converse does not hold in general but it holds when D_1 is reduced:

Theorem 2. *If $D_1 \preceq D_2$ and D_1 is reduced, then $D_1 \subseteq D_2$.*

This theorem is proven using the following useful property of reduced descriptions:

Lemma 3. *Let D be a consistent, reduced description. Then there exists a life satisfying D in which every formula of D is realized.*

3.4 Describing Behaviors

We define the following general notion:

Definition 5. *A* behavior *over a schema* $\mathbf{S}$ *is a set of lives over* $\mathbf{S}$.

One should generally think of a behavior as the set of lives that can be generated through some database application. We will soon define a language for specifying such applications. Here, we already introduce some language-independent notions.

We have already studied lots of behaviors, since, for any description D, the set $\mathit{lives}(D)$ is a behavior. The goal is to describe some given behavior B using the behavior of a description D. We can use the standard terminology to compare the behavior of D to B:

- D is *sound* for B if $lives(D) \subseteq B$.
- D is *complete* for B if $B \subseteq lives(D)$.

The descriptions studied in this paper are somehow coarse-grained, as they describe only the growth or shrinking of relations, and thus allow quite arbitrary transitions. Hence, a given behavior will typically not have nontrivial sound descriptions. In this paper we will mainly be interested in obtaining a complete description instead. Moreover, we would like it to be as tight as possible. Formally, we say that D is a *minimal* complete description for B if D is complete for B and no strict subset of D is complete for B.

We can see that the object of our quest always exists and is unique:

Proposition 4. *Every behavior B has a unique minimal complete description; this description is reduced.*

In Sect. 5 we will investigate the effective computability of minimal complete descriptions for the behaviors of database applications, introduced next.

4 A Formal Model of Database Applications

In this section we define *database applications*, which are finite collections of *procedures*. Each procedure will consist of a parameter declaration and an update program. We will use the formal model of update programs proposed in our earlier work with Ameloot [4]. In that proposal, update programs are built from an arbitrary query language. In the present work, we will use relational algebra as the query language.

4.1 Relational Algebra Expressions and Update Programs

To fix notation, we recall the relational algebra as we will use it in this paper. We then recall the formal model of update programs.

Let $\mathbf{S}$ be a database schema. The relational algebra expressions over $\mathbf{S}$ (abbreviated rae) are inductively built as follows.

- Every relation name $R \in \mathbf{S}$ is an rae with the arity given by the schema.
- For any constant $c \in \mathbf{dom}$, we allow $\{c\}$ as an rae of arity one.
- If e is an rae of arity m, and $i, j \in \{1, \ldots, m\}$, then $\sigma_{i=j}(e)$ is an rae of arity m.
- If e is an rae of arity m, and $i_1, \ldots, i_k \in \{1, \ldots, m\}$, then $\pi_{i_1,\ldots,i_k}(e)$ is an rae of arity k.
- If e_1 and e_2 are raes of the same arity m, then $e_1 \cup e_2$, as well as $e_1 \cap e_2$ and $e_1 - e_2$ are raes of arity m.
- If e_1 and e_2 are raes of arities m and k respectively, then $e_1 \times e_2$ is an rae of arity $m + k$.

Given an instance I of $\mathbf{S}$, an rae e of arity m evaluates to an m-ary relation $e(I)$ as follows. The semantics of a relation name R is given by the instance. The semantics of $\cup$, $\cap$ and $-$ is given by the corresponding set operations. The semantics of $\times$ is Cartesian product. The semantics of $\{c\}$ is the singleton unary relation $\{(c)\}$. For a relation r of the appropriate arity, the projection $\pi_{i_1,\ldots,i_k}(r)$ equals $\{(a_{i_1},\ldots,a_{i_k}) \mid \bar{a} \in r\}$ and the selection $\sigma_{i=j}(r)$ equals $\{\bar{a} \in r \mid a_i = a_j\}$.

Remark 6. We will feel free to use abbreviations that are definable by raes, for example, $\emptyset$, inclusions, boolean connectives, if-then-else, quantifiers, relational calculus formulas, etc. [2]. □

We define three update operations:

Insert For a relation name R and an rae e of the same arity as R, the operation $\mathsf{insert}_R(e)$, applied to an instance I, results in the instance J that is the same as I except that $J(R) = I(R) \cup e(I)$.

Delete For a relation name R and an rae e of the same arity as R, the operation $\mathsf{delete}_R(e)$, applied to an instance I, results in the instance J that is the same as I except that $J(R) = I(R) - e(I)$.

Modify For a relation name R of arity k and an rae e of arity $2k$, the operation $\mathsf{modify}_R(e)$, applied to an instance I, is defined as follows. A $2k$-ary relation can be used as a function from k-tuples to k-tuples, on condition that it does not contain two tuples of the form (t, t') and (t, t'') where t, t' and t'' are k-tuples and $t' \neq t''$. If $e(I)$ is not a function in this way, the operation is undefined. Otherwise, it results in the instance J that is the same as I except that

$$J(R) = \{t \in I(R) \mid \neg\exists t' : (t, t') \in e(I)\} \cup \{t' \mid \exists t \in I(R) : (t, t') \in e(I)\}.$$

The modify operation formalizes the update statement in SQL, which requires that only scalar subqueries (i.e., functions) can be used in the assignment clauses. We say that a modify operation as above is *always defined* if $e(I)$ is a function on every instance I.

We next define *update programs* as follows.

- Every update operation (insert, delete, modify) is a program.
- If Π_1 and Π_2 are programs, then $\Pi_1; \Pi_2$ (sequential composition) is also a program.
- If e is an rae and Π_1 and Π_2 are programs, then '$\mathsf{if}\ e\ \mathsf{then}\ \Pi_1\ \mathsf{else}\ \Pi_2\ \mathsf{endif}$' is a program.
- If e is an rae and Π is a program, then '$\mathsf{while}\ e\ \mathsf{do}\ \Pi\ \mathsf{enddo}$' is a program.

The result of executing a program Π on an instance I results in an instance $\Pi(I)$ defined in the obvious way. The meaning of $\mathsf{if}\ e$ is $\mathsf{if}\ e \neq \emptyset$ and similarly for while. If a modify operation has an undefined result, or a while loop does not terminate, the result of the program is not defined.

Remark 7. A no-op can be simulated by $\mathsf{insert}_R(\emptyset)$, so $\mathsf{if}\ e \neq \emptyset\ \mathsf{then}\ \Pi\ \mathsf{endif}$ without else can be simulated by $\mathsf{if}\ e \neq \emptyset\ \mathsf{then}\ \Pi\ \mathsf{else}$ no-op endif.

4.2 Database Applications

We are now ready to define procedures, applications, and their behavior.

A *procedure* over a schema $\mathbf{S}$ is a triple $(Param, k, \Pi)$, where $Param$ is a relation name not in $\mathbf{S}$ and Π is an update program over $\mathbf{S}$ extended with relation name $Param$ of arity k. We refer to k as the *parameter arity* of the procedure. A *procedure call* is of the form $P(\bar{a})$, where P is a procedure and $\bar{a}$ is a k-tuple of values in **dom**, with k the parameter arity of P. The result of $P(\bar{a})$ in an instance I of $\mathbf{S}$ is defined to be the result of executing Π on I', where I' is the expansion of I by setting relation $Param$ to the singleton $\{\bar{a}\}$.

A *database application* (or *application* for short) over $\mathbf{S}$ is a finite nonempty set A of procedures over $\mathbf{S}$. A life l over $\mathbf{S}$ is said to be a life *of application* A if for every transition (I, J) in l, we have that J is the result of a procedure call $P(\bar{a})$ applied to I, where $P \in A$. We refer to the set of all lives of A as the *behavior* of A, denoted by $lives(A)$.

Remark 8. Since the result of a program may be undefined because of ill-defined modify operations or infinite while-loops, it is possible in extreme cases that an application does not have any life. An example is the application with a single procedure with the following program:

```
while true do
  if R = ∅ then insert_R({c}) else delete_R({c}) endif
enddo.
```

Another example uses as program instead the operation $\mathsf{modify}_R(\{c\}\times\{c, d\})$, for different constants c and d, as this operation is always undefined.

Less extremely, it is possible that a life in the behavior of an application A ends in an instance on which all procedure calls are undefined. Then this life is not continued in any other life in the behavior of A.

Example 5. Consider a vacation house reservation application over a schema with relations House(hid, location, price); Customer(cid, name); Voucher(vid, cid, amount); and Stay(cid, hid, week). We have the following procedures. For the sake of readability, we take some liberty with the formalism.

Procedure *newCustomer* with Param(cid, name, vid, amount):

```
if π1(Param) ⊈ π1(Customer) ∧ π3(Param) ⊈ π1(Voucher)
then insert_Customer(π1,2(Param));
     insert_Voucher(π3,1,4(Param))
endif
```

Procedure *buyVoucher* with Param(vid, cid, amount) inserts a new voucher for an existing customer. Procedure *newHouse* with Param(hid, location, price) inserts a new house. The programs are similar to the one above.

Procedure *reserve* with Param(cid, hid, week, vid):

$$\begin{array}{l}\mathsf{if}\ \pi_{1,4}(\text{Param}) \subseteq \pi_{2,1}(\text{Voucher}) \\ \quad \wedge\, \pi_{2,7}\,\sigma_{4=5}(\text{Param} \times \text{Voucher}) \subseteq \pi_{1,3}(\text{House}) \\ \quad \wedge\, \pi_{2,3}(\text{Param}) \nsubseteq \pi_{2,3}(\text{Stay}) \\ \mathsf{then}\ \mathsf{insert}_{\text{Stay}}(\pi_{1,2,3}(\text{Param})); \\ \qquad \mathsf{delete}_{\text{Voucher}}(\pi_{2,3,4}\,\sigma_{1=2}(\pi_4(\text{Param}) \times \text{Voucher})) \\ \mathsf{endif}\end{array}$$

Our final two procedures have an update program consisting of a single operation. Procedure *changePrice* with Param(hid, price):

$$\mathsf{modify}_{\text{House}}(\pi_{1,2,3,1,2,5}\,\sigma_{1=4}(\text{House} \times \text{Param})).$$

Procedure *reset*, without a parameter:

$$\mathsf{delete}_{\text{Stay}}(\text{Stay}).$$

5 Descriptions of Applications

As seen in Proposition 4, every behavior has a minimal complete description. In this section we investigate whether the minimal complete description of the behavior of a given application is effectively computable.

Table 1. Minimal complete description of the behavior of the vacation house reservation application.

House	Customer	Voucher	Stay	procedure
=	=	=	=	*newCustomer*
=	↗	↗	=	*newCustomer*
=	=	=	=	*buyVoucher*
=	=	↗	=	*buyVoucher*
=	=	=	=	*newHouse*
↗	=	=	=	*newHouse*
=	=	=	=	*reserve*
=	=	↘	↗	*reserve*
=	=	=	=	*changePrice*
=	=	=	=	*reset*
=	=	=	↘	*reset*

Example 6. Recall the application from Example 5. The minimal complete description of its behavior is shown, in tabular form, in Table 1. The table also shows which procedures can give rise to which formulas. The first line of the table, which represents the formula $\{\text{House}^{=}, \text{Customer}^{=}, \text{Voucher}^{=}, \text{Stay}^{=}\}$, reflects the possibility that none of the relations change size. This happens, for example,

when *newCustomer* is called with an existing cid, due to the if-condition in the program. It happens also each time we call *changePrice*. Indeed, the expression in the modify operation always returns a function, because relation Param is always a singleton and attribute hid is always a key for relation House in every instance of every life. This key constraint holds because procedure *newHouse* will only insert when given a new hid. □

Below we will show undecidability results, but we will also identify decidable cases. First, we investigate the inference of individual size primitives. In Sect. 5.2 we turn to the inference of formulas. We will also consider an extension of our formal model with value invention.

5.1 Inferring Size Primitives

Recall that a life l is said to realize a formula F if some transition in l satisfies F. Similarly, we say that l realizes a size primitive p if some transition in l satisfies p. We now formally define the following decision problem:

Problem: GROW.
Input: An application A over a database schema $\mathbf{S}$, and a relation name R in $\mathbf{S}$.
Decide: There exists a life of A that realizes $R^{\nearrow}$.

Similarly we can define the problems SHRINK and SAMESIZE which are about realizing $R^{\searrow}$ and $R^{=}$, respectively.

Deciding GROW

Example 7. Consider an application A consisting of a single procedure P over two binary relations R_1, R_2, with parameter arity two, with e an rae over R_2 alone:

if e then $\mathsf{insert}_{R_1}(Param)$ else $\mathsf{insert}_{R_2}(Param)$ endif

We see that there exists a life of A where R_1 grows if and only if e is satisfiable, i.e., nonempty on some instance. Indeed, if e is satisfiable, let I be a minimal instance on which e is nonempty. Let $I(R_2) = \{t_1, \ldots, t_n\}$. We can make n calls $P(t_1)$, ..., $P(t_n)$, which generates a life ending in the instance J where $J(R_1)$ is empty and $J(R_2) = \{t_1, \ldots, t_n\}$. Making one more call $P(t)$ will insert t into R_1, causing R_1 to grow. □

Since satisfiability of relational algebra expressions over a binary relation is undecidable [2,5], the above example shows:

Theorem 3. *Problem* GROW *is undecidable, even in restriction to applications with a single procedure that does not use while-loops.*

Remark 9. Some variations of the application A seen in Example 7 are possible. We can avoid if-then-else in the update program but use two statements instead:

$\mathsf{insert}_{R_1}(\text{if } e \text{ then } \mathit{Param} \text{ else } \emptyset)$;
$\mathsf{insert}_{R_2}(\text{if } e = \emptyset \text{ then } \mathit{Param} \text{ else } \emptyset)$.

Here the if-then-else constructs are not those of the update language, but are expressed in relational algebra.

We can also use two procedures, with a single operation each:

Procedure P_1: $\mathsf{insert}_{R_1}(\text{if } e \text{ then } \mathit{Param} \text{ else } \emptyset)$.
Procedure P_2: $\mathsf{insert}_{R_2}(\mathit{Param})$.

We see in both of these variations that there exists a life of the application where R_1 grows, if and only if e is satisfiable. □

In contrast, we can identify the following decidable case:

Theorem 4. *Problem* GROW *is decidable for applications over a single relation.*

The proof exploits the C-genericity of relational algebra expressions to test the procedure from the empty instance using only a finite set of parameter tuples.

We can determine the precise complexity of the problem identified in the above theorem for a class of applications that we call *single-statement*. Let us define single-statement programs as those built from update operations using only the if-then-else construction. So, sequential composition and while-loops are not allowed. An application is called single-statement if every procedure has a single-statement program.

Proposition 5. *Problem* GROW *for single-statement applications over a single relation name is PSPACE-complete.*

The proof leverages the PSPACE combined complexity of relational algebra evaluation for the upper bound, and leverages the PSPACE-hard expression complexity of relational algebra evaluation for the lower bound.

Remark 10. Determining the complexity of executing general update programs is an interesting topic for further research. In the above we restrict to single-statement programs so that we can rely on known results on the complexity of evaluating relational algebra. Note that, using sequential composition, an update program can encode a relational algebra expression of exponential size. For example, consider the following program over a binary relation name R:

$\mathsf{insert}_T(\pi_{1,4}\,\sigma_{2=3}(R \times R))$;
$\mathsf{insert}_T(\pi_{1,4}\,\sigma_{2=3}(T \times T))$;
$\vdots$
$\mathsf{insert}_T(\pi_{1,4}\,\sigma_{2=3}(T \times T)$.

Let n be the number of operations. Thinking of relation R as a directed graph, this program computes in T all pairs (a, b) such that there is a walk from a to b of length 2^i for some $i \in \{1, \ldots, n\}$.

Deciding SHRINK. Call an application A *insert-only* if for every relation name R, if an operation delete_R or modify_R appears anywhere in the application, then the application does not contain any operation insert_R. Of course this means that for each operation delete_R or modify_R in the program, relation R will remain empty since it is never inserted into. As a consequence, no R can ever shrink. We conclude that Problem SHRINK is trivially decidable for insert-only applications.

Outside of insert-only applications, Problem SHRINK is typically undecidable, as shown by the following Theorem. The proof requires some ingenuity in constructing detailed programs that, despite the imposed syntactical limitations, can exactly simulate undecidable problems about relational algebra expressions.

Theorem 5. *Problem* SHRINK *is undecidable, even in restriction to applications over a single binary relation name, in the following classes:*

1. *A single procedure consisting of a single if-then-else statement, with a single insert operation as the then-case, and a single delete operation as the else-case.*
2. *A single procedure consisting of a single if-then-else statement, with a single insert operation as the then-case, and a single, always-defined, modify operation as the else-case.*
3. *Two procedures, one consisting of a single insert statement, and one consisting of a single delete statement.*
4. *Two procedures, one consisting of a single insert statement, and one consisting of a single, always-defined, modify statement.*

Deciding SAMESIZE. Recall the class of insert-only applications defined in the previous subsection. We saw there that such applications can never realize a shrink primitive. We can also define the notion of insert-only for individual programs, in a similar way. Naturally, a procedure is then called insert-only if its program is. For the same-size primitive, we now observe the following behavior. We omit while-loops and modify operations as these may have undefined results (compare Remark 8).

Proposition 6. *Let A be an application with at least one insert-only procedure without while-loops and without modify operations, and let R be a relation name from the database schema. Then there always exists a life of A that realizes $R^{=}$.*

The idea behind the proof is that calling the insert-only procedure repeatedly with the same parameter must necessarily converge.

As soon as we deviate even slightly from the case described in the above proposition, the problem becomes undecidable, as shown next. The first two cases of the following theorem are the same as in Theorem 5, but quite different reductions had to be invented to prove them.

Theorem 6. *Problem* SAMESIZE *is undecidable, even in restriction to applications over a single binary relation name, with a single procedure, in the following classes:*

1. *A single if-then-else statement, with a single insert operation as the then-case, and a single delete operation as the else-case.*
2. *A single if-then-else statement, with a single insert operation as the then-case, and a single, always-defined, modify operation as the else-case.*
3. *A single if-then-else statement, with a single insert operation as the then-case, and a while-loop with a no-op body as the else-case.*[2]

5.2 Computing the Minimal Complete Description

Ideally we would like to be able to compute the minimal complete description of a given application. Since, over any given database schema, the number of possible formulas is finite, this amounts to solving the following problem.

Problem: FORMULA
Input: An application A over a database schema $\mathbf{S}$, and a formula F over $\mathbf{S}$.
Decide: There exists a life of A that realizes F.

Problem FORMULA is a more general problem than the problems GROW, SHRINK and SAMESIZE. Thus, FORMULA is undecidable and can only be decidable for classes of inputs where the three more specific problems are decidable. Given the results of the previous subsection, this results in the class of insert-only applications over a single relation name, without while-loops. When there is only one relation name, there is no difference between inferring formulas and inferring size primitives. We conclude:

Theorem 7. *For insert-only applications over a single relation name, without while-loops, the minimal complete description is effectively computable.*

Remark 11. We can extend the above result to "partitioned" applications, where the procedures can be partitioned into groups so that all procedures in each group work on a single relation name, and these relation names are different for different groups. For example, assume application A is partitioned into A_1 and A_2, on relation names R_1 and R_2 respectively. If D_1 (D_2) is the minimal complete description of A_1 (A_2), then the minimal complete description of A equals

$$\{F \cup \{R_2^=\} \mid F \in D_1\} \cup \{F \cup \{R_1^=\} \mid F \in D_2\}.$$

5.3 Programs with Value Invention

In practice, procedures often use an auto-increment feature that generates fresh identifiers. Relational algebra programs with value invention are known to be highly expressive [2]. Indeed, we next show that Proposition 6 no longer holds if this feature is introduced.

We formalize auto-increment by extending the notion of procedure from a triple $(Param, k, \Pi)$ to a 4-tuple $P = (Param, k, v, \Pi)$, with $v \in \{0, \ldots, k\}$. The

[2] For no-op, see Remark 7.

idea is that in a call, the last v parameters will be invented values. Formally, the parameter tuple $\bar{a}$ given in the call now has arity $k-v$ instead of k. When calling $P(\bar{a})$ on instance I, the program is executed on the expansion of I where relation *Param* is now instantiated with $\{(\bar{a}, \bar{z})\}$, with $\bar{z}$ a v-tuple of domain elements outside the active domain of I. Accordingly, we say that P is auto-incrementing if $v \neq 0$.

We show:

Proposition 7. *Problem* SameSize *is undecidable for applications over a single ternary relation name R, with a single auto-incrementing procedure, having a single insert operation as its program.*

In this proof, although all other hypotheses of Proposition 6 hold, the auto-increment can introduce new elements in the active domain, which allows us to simulate arbitrary computations by insertions only.

Remark 12. Theorem 4 continues to hold in the presence of auto-increment; the proof remains exactly the same. Indeed, the proof focuses on the very first transition, which starts from the empty instance, so the auto-incrementing is irrelevant.

6 Conclusion

We have proposed the novel perspective of analyzing the database evolution through the lens of size changes. We see many directions for further research.

One natural direction is to revisit the decidability and undecidability results (Sect. 5) when we restrict the language used to express the queries embedded in the update language. In this work, we have used relational algebra, which has an undecidable satisfiability problem. Omitting the difference operator leads to the class of unions of conjunctive queries, which have a decidable implication problem [2].

In our current analysis of database applications, we do not consider any constraints on the order in which procedures can be called. In practice such constraints will often be naturally governed by a given workflow or process model.

Also, in our current approach, descriptions only look at what can happen in individual transitions of a life. A more fine-grained approach could be to label every transition of a life with its proper size formula. This yields a view of lives as strings of size formulas. Since the set of possible size formulas over a given database schema forms a finite alphabet, this allows us to combine our size-change perspective with standard temporal logics for expressing properties of strings and traces.

Finally, in Example 6, we saw that the analysis of behaviors can hinge on constraints (e.g., keys) on reachable database states that are implied by the application. Seminal work in this direction was done by Abiteboul and Vianu [3]. We feel this line of research still has a lot of promise and potential.

References

1. Abdulla, P., Aiswarya, C., Atig, M., Montali, M., Rezine, O.: Recency-bounded verification of dynamic database-driven systems. In: Proceedings 35th ACM Symposium on Principles of Database Systems, pp. 195–210 (2016)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
3. Abiteboul, S., Vianu, V.: A transaction-based approach to relational database specification. J. ACM **36**(4), 758–789 (1989)
4. Ameloot, T., Van den Bussche, J., Waller, E.: On the expressive power of update primitives. In: Proceedings 32nd ACM Symposium on Principles of Database Systems, pp. 139–150 (2013)
5. Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. Springer (1997)
6. Calvanese, D., De Giacomo, G., Montali, M.: Foundations of data-aware process analysis: a database theory perspective. In: Proceedings 32nd ACM Symposium on Principles of Database Systems, pp. 1–12 (2013)
7. Deutsch, A., Hull, R., Li, Y., Vianu, V.: Automatic verification of database-centric systems. ACM SIGLOG News **5**(2), 37–56 (2018)
8. Deutsch, A., Sui, L., Vianu, V.: Specification and verification of data-driven Web applications. J. Comput. Syst. Sci. **73**(3), 442–474 (2007)
9. Itzhaky, S., et al.: On the automated verification of web applications with embedded SQL. In: Benedikt, M., Orsi, G. (eds.) Proceedings 20th International Conference on Database Theory. LIPIcs, vol. 68, pp. 16:1–16:18. Schloss Dagstuhl–Leibniz Center for Informatics (2017)
10. Silberschatz, A., Korth, H., Sudarshan, S.: Database System Concepts. Mc Graw-Hill, 7th edn. (2019)
11. Spielmann, M.: Verification of relational transducers for electronic commerce. J. Comput. Syst. Sci. **66**(1), 40–65 (2003)