# On the expressive power of update primitives

Tom J. Ameloot Jan Van den Bussche Emmanuel Waller

## Abstract

The SQL standard offers three primitive operations (insert, delete, and update which is here called modify) to update a relation based on a generic query. This paper compares the expressiveness of programs composed of these three operations, with the general notion of update that simply replaces the content of the relation by the result of a query. It turns out that replacing cannot be expressed in terms of insertions, deletions, and modifications, and neither can modifications be expressed in terms of insertions and deletions. The expressive power gained by if-then-else control flow in programs is investigated as well. Different ways to perform replacing are discussed: using a temporary variable; using the new SQL merge operation; using SQL's data change delta tables; or using queries involving object creation or arithmetic. Finally the paper investigates the power of alternating the different primitives. For example, an insertion followed by a modification cannot always be expressed as a modification followed by an insertion.

## 1 Introduction

The three basic update primitives for relational databases are the insertion of a tuple in a relation; the deletion of a tuple from a relation; and the modification of the components of a tuple in a relation. These three primitives are so simple, so natural and so commonplace that they are hardly ever questioned, apart perhaps from the folk wisdom that instead of modifying $t$ to $t'$, one can equivalently insert $t'$ and delete $t$.

The database query language SQL, however, provides much more powerful versions of the three basic update operations. An SQL insert operation allows to insert an entire set of tuples, given by a query applied to the current instance. Likewise an SQL delete operation allows to delete an entire set of tuples, again determined by a condition that can be an arbitrarily complex boolean query. For example, acting on a relation R(A,B) in SQL we can write `delete from R R0 where exists (select R1.B from R R1 where R1.B=R0.A)`.

Also an SQL update operation allows to modify a set of tuples determined by a query condition, and moreover, the new values of the modified tuples can again be determined by queries as well, which are applied to the instance before the update. These queries must return single values (the so-called "scalar subqueries"). A classical example is to modify the salary of some employee to the current average salary, but scalar subqueries are not restricted to aggregate functions. For example, in SQL we can write `update R R0 set R0.B = (select R1.B from R R1 where R1.A=R0.B)`. This will succeed on any instance satisfying the constraint that for every tuple $(a_0, b_0)$ in $R$ there exists exactly one tuple $(a_1, b_1)$ in $R$ with $a_1 = b_0$. By using a more complicated query, we can make the update succeed on any instance, leaving tuples $(a_0, b_0)$ that violate the constraint unchanged.

The goal of the present paper is to understand better the expressive power of these query-based insertions, deletions and modifications. For instance, can we still express a modification by an insert followed by a delete? At first this may appear to be the case. For a simplistic example, consider again a relation R(A,B), and a typical modification like `update R set B=0 where A=5`. Assuming we are satisfied with relations as sets, as opposed to SQL's relations as bags, we can express this modification by `insert into R (select A,0 from R where A=5)` followed by `delete from R where A=5 and not B=0`. However, this is not always so easy; the reader is invited to try to express `update R set A=B, B=A`, which simply swaps the two columns, by a sequence of insertions and deletions acting solely on $R$.

The last qualification is important: if we can use a temporary relation $S$ as scratch space, then quite trivially we can express *any* update using only insertions and deletions. More specifically, for any query $Q(R)$ that maps instances of $R$ to instances of $R$, we may consider the *replacement* update that replaces the content of $R$ by $Q(R)$.[1] Any such replacement can be expressed by the following crude procedure: erase $S$; insert $Q(R)$ in $S$; erase $R$; and insert all of $S$ in $R$. Interestingly, we will see that this procedure can also be mimicked using SQL:2011's new "data change delta tables" [29, 33]. Furthermore, the procedure can also be mimicked when the query language is powerful enough and allows the introduction of new data elements; or when the data elements in the relation are numbers and the query language can do arithmetic.

Nevertheless, achieving a desired update "in place" may be preferred over ways that copy information around, for instance for reasons of efficiency. So we should better un-

[1] This general notion of replacement should not be confused with MySQL's 'replace' operation [28], which behaves like an "upsert": an insert that behaves like a modify for insertions that violate some declared key constraint.

derstand the *intrinsic* expressive power of the basic update primitives, in the absence of temporary relations. The ideal framework for such a study is given by the *generic queries* from the classical theory of database query languages [2, 5, 7, 14, 22]. Such queries do not interpret data elements as numbers, and neither do they introduce new data elements, but apart from that they cover exactly all data manipulations on the level of the logical structure of the relations in the database. In this framework every update to a relation can be modeled as a replacement update, defined by some generic query $Q$.

We consider a simple update language, called $\mathcal{UL}$, where programs are built from insertions, deletions and modifications using sequential composition and if-then-else statements. We will show that modification is primitive in $\mathcal{UL}$: there exist modifications that are not expressible by any program not using modification. An example is the modification that swaps the two columns of $R$, already mentioned above. (Insertion and deletion are likewise primitive, but this is obvious.) Moreover, $\mathcal{UL}$ is not update-complete: there exist simple updates that are not expressible by any program. This result holds even though arbitrary queries can be used in the update operations. An example of an inexpressible update is the replacement of $R$ by its complement (with respect to its active domain).

Interestingly, the latest SQL:2011 standard [29, 33] has a *merge* operation that allows to combine insertions and deletions (and modifications as well) in a single operation.[2] We will provide a formalization of this operation and will show that any replacement can be expressed by a single merge. In this sense, our results show that the addition of merge has strictly increased the power of SQL's repertoire of update operations.

We also investigate how the if-then-else construct influences the expressive power of update programs. We show that in general it does, not only in the full language, but also in every fragment (obtained by omitting one or two update primitives). These results hold even though if-then-else may well be expressible by the queries used inside the individual update operations.

Finally we investigate the expressive power of alternating the update primitives. For database queries expressed in first-order logic, it is well known that using more alternations between existential and universal quantifiers allows the expression of more queries [15]. In our setting, we show that an insertion followed by a modification is not always expressible by a modification followed by an insertion. The converse separation holds as well, and we have similar separations for combinations of insertions and deletions, and of deletions and modifications. These results can be likened to the separation of the levels $\Sigma_1$ and $\Pi_1$ in the above-mentioned quantifier alternation hierarchy; but this analogy should not be taken too literally. Indeed, as in all our results, the generic queries used inside the update operations can be arbitrarily powerful. Since we have solved only the first level, the hierarchy for higher levels of alternation of update primitives remains to be further explored.

Most of the inexpressibility proofs amount to showing that, when trying to express some update by an unsuitable program, some loss of information cannot be prevented. This points at the fundamental weakness of in-place updating. The property that generic queries do not distinguish between isomorphic instances, or subinstances, plays an essential role in our arguments.

To conclude this introduction we would like to remark that efficiency, mentioned above as a potential motivation, is not our only motivation, as indeed we do not give any concrete results regarding efficiency. We mainly believe that the simplicity of the question of in-place updates calls for a thorough foundational understanding. Moreover, update operations are ubiquitous and, as mentioned above, as late as 2011 the SQL standard has made two fundamental additions to SQL's update features. This shows that there is a still current interest in powerful in-place updating in practice. We hope our work may serve in part as some theoretical justification of these additions to the standard.

This paper is organized as follows. Section 2 discusses related work. Section 3 defines the update language $\mathcal{UL}$ formally. Section 4 presents the results on the expressive power of the language and its fragments. Section 5 presents the results on alternation. Section 6 discusses the different ways in which one can express general replacement updates. Section 7 concludes with a discussion of open problems and topics for further research.

## 2 Related work

The authoritative reference on the expressive power of update languages is still Abiteboul and Vianu's work on the language TL and its variants [5, 6]. The present paper is complementary in scope to that work. Indeed, in TL, temporary relations are taken for granted, so the questions investigated in the present paper were not considered. For instance, in TL, modifications are superfluous. Abiteboul and Vianu also invented a technique of versioning using value invention (the introduction of new data elements). By this technique, in a setting with temporary relations and where input relations need not be preserved (altogether different from our setting), even deletions can become superfluous [6]. In Section 6 we will show a different application of the value invention technique, to simulate replacement updates using just in-place insertions and deletions.

An even earlier seminal paper by Abiteboul and Vianu is that on relational transactions [4]. In that work, programs that are sequential compositions of insertions, dele-

[2] Earlier editions of SQL already had a version of merge, but did not allow to combine insertions and deletions.

tions and modifications are considered, very much as in the present paper. However, the queries inside the update operations in a relational transaction are limited to selections expressed by conjunctions of equalities and nonequalities between attributes and constants. In the present paper, formalizing the update operations of SQL, we allow at least first-order queries, or even arbitrary generic queries inside update operations.

In the present paper we work in the relational data model and are inspired by SQL, but it should be noted that over the past decade there has also been much interest in providing facilities to update XML data in the context of XQuery [30, 10, 32]. It should be interesting to extend our work to that context; indeed some work on the expressive power of the XQuery-based update primitives has already been reported by the Antwerp school [21].

In general there is a large literature concerned with updates, investigating topics such as equivalence [23], order-independence [25, 8], commutativity [17, 19], type checking [11] and independence [26, 12]; we can only give a few references. The recent work cited above is mostly in the setting of XML [13]; updates in the relational model are the topic in Abiteboul's 1988 invited paper [1]. We also recall the rather large interest in declarative specification of updates that existed in the 1990s; we only cite two reference volumes [27, 18]. Finally we mention the challenging and ever ongoing topic of belief revision and knowledge update in the field of Artificial Intelligence.

# 3 Updates

In this section we formalize the three basic update primitives (insertion, deletion, and modification) and also introduce the general primitive of replacement. We then define the simple update language considered in this paper.

## 3.1 Preliminaries

We recall some basic notions and terminology from the theory of relational databases. 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. The set of all $k$-ary relations is denoted by $Rel(k)$. A *database schema* is a finite set $\mathbf{S}$ of *relation names* where every relation name has an associated arity (a natural number). An *instance* $I$ of $\mathbf{S}$ 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 instances of $\mathbf{S}$ is denoted by $inst(\mathbf{S})$.

**Generic queries and updates** We next define queries and updates as two special kinds of database transformations as defined in general by Abiteboul and Vianu [5]. Recall that a *permutation* of **dom** is a bijection from **dom** to **dom**. A permutation can also be applied to a relation or to an instance, simply by applying the permutation to all appearances of data elements in the relation or instance.[3] When a permutation $\rho$ is the identity on some given subset $C \subseteq \mathbf{dom}$, then $\rho$ is also called a $C$-permutation.

Let $k$ be a natural number and let $\mathbf{S}$ be a database schema.

- A $k$-ary *query* over $\mathbf{S}$ is a mapping $q : inst(\mathbf{S}) \to Rel(k)$ for which there exists a finite set $C \subseteq \mathbf{dom}$ such that for every $C$-permutation $\rho$ of **dom**, we have $q(\rho(I)) = \rho(q(I))$ for all instances $I$. We say that $q$ is *$C$-generic*.
- An *update* over $\mathbf{S}$ is a partial mapping $u : inst(\mathbf{S}) \to inst(\mathbf{S})$ that is again $C$-generic for some finite set $C \subseteq \mathbf{dom}$, i.e., for every $C$-permutation $\rho$ and every instance $I$, we have $u(I)$ is defined iff $u(\rho(I))$ is defined, and if so then $u(\rho(I)) = \rho(u(I))$.

The intuition behind $C$-genericity is that the query or update may interpret the constants in $C$ specially, but otherwise treats all data elements generically, in the typical database fashion of set-oriented bulk data processing [2, 7, 14, 22]. For example, the SQL query `select * from R where A=5` is $\{5\}$-generic, and the SQL update `update R set B=0 where A=5` is $\{0, 5\}$-generic.

We assume familiarity with first-order logic, relational algebra, and relational calculus as basic languages for expressing queries [2].

We also recall that a nullary query can be likened to a logic formula without free variables, thus merely returning a boolean value true (the nonempty nullary relation) or false (the empty nullary relation) on every instance. Hence nullary queries are also called *boolean* queries. Boolean queries will be used as the conditions in the if-then-else statements of our update language.

*Remark* 3.1. We allow updates to be partial mappings because in general, modification updates may be undefined on some instances as we will see in the next section. We assume queries to be total for simplicity; our results do not depend on this assumption. Also, one normally requires queries and updates to be computable. We omit this requirement simply because we do not need it. We assure the reader that none of our results relies on the use of noncomputable queries.

## 3.2 Update operations

**Replace** The most general update operation is replace, defined as follows. Let $\mathbf{S}$ be a database schema and $R \in \mathbf{S}$

[3] Formally, if $t = (a_1, \ldots, a_k)$ is a tuple and $\rho$ is a permutation, then $\rho(t)$ equals the tuple $(\rho(a_1), \ldots, \rho(a_k))$. If $r$ is a relation then $\rho(r)$ equals the relation $\{\rho(t) \mid t \in r\}$. Finally, if $I$ is an instance then $\rho(I)$ equals the instance given by $\rho(I)(R) = \rho(I(R))$ for each relation name $R$.

a relation name of arity $k$. Let $q$ be a $k$-ary query over $\mathbf{S}$. Then $\mathsf{replace}_R(q)$ is the update over $\mathbf{S}$ defined as follows. For any instance $I$ of $\mathbf{S}$, we have $\mathsf{replace}_R(q)(I)(R) = q(I)$, and $\mathsf{replace}_R(q)(I)(S) = I(S)$ for each $S \neq R$. So, the content of relation $R$ is simply replaced by the result of the query $q$.

**Insert** The update $\mathsf{insert}_R(q)$ can now be defined as $\mathsf{replace}_R(q')$, where $q'$ is the query defined by $q'(I) = I(R) \cup q(I)$.

**Delete** The update $\mathsf{delete}_R(q)$ can now be defined as $\mathsf{replace}_R(q')$, where $q'$ is the query defined by $q'(I) = I(R) - q(I)$ (set difference).

**Modify** For a modification of a relation $R$ of arity $k$, we need a query $q$ of arity $2k$ rather than just $k$. The intuition is that $q$ returns pairs of tuples $(t, t')$ where $t'$ is the modified version of $t$. To avoid ambiguity we require $q$ to return a $2k$-ary relation that is a *function* on $k$-tuples, i.e., that does not contain two tuples of the form $(t, t')$ and $(t, t'')$ where $t$, $t'$ and $t''$ are $k$-tuples and $t' \neq t''$. This formalizes the requirement in SQL that only scalar subqueries can be used in the assignment clause of an SQL update statement.

Formally, we define the update $\mathsf{modify}_R(q)$ to be defined on an instance $I$ only if $q(I)$ is a function. In that case $\mathsf{modify}_R(q)(I)$ is defined to be equal to $\mathsf{replace}_R(q')(I)$, where $q'$ is the query defined by

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

So, the tuples from relation $R$ that are not mentioned in the result of $q$ are left untouched, and the other tuples are modified. We feel this definition most elegantly formalizes the use of queries in an SQL update statement, by bundling the query in the where-clause together with the queries used in the assignment clause, all in a single $2k$-ary query.

*Example* 3.2. For a binary relation $R(A, B)$ the SQL statement update R set B=0 where A=5 can be modeled as $\mathsf{modify}_R(q)$, where $q$ is the query $\{(5, y, 5, 0) \mid R(5, y)\}$ (expressed in first-order logic). Likewise, the SQL statement update R set B=A, A=B is modeled as $\mathsf{modify}_R(q)$, where $q$ is the query $\{(x, y, y, x) \mid R(x, y)\}$. Now recall the SQL update from the Introduction, update R R0 set R0.B = (select R1.B from R R1 where R1.A=R0.B). It would not be quite correct to model this as $\mathsf{modify}_R(q)$ where $q$ is the query $\{(x, y, x, z) \mid R(x, y) \wedge R(y, z)\}$. Indeed, by our above-defined semantics, that update leaves any tuples in $q_{\text{undef}} := \{(x, y) \mid R(x, y) \wedge \neg\exists z\, R(y, z)\}$ untouched, whereas the SQL update is well defined only if $q_{\text{undef}}$ is empty. Hence the strictly correct way to model the SQL update is to use for $q$ the query $\{(x, y, x, z) \mid q_{\text{undef}} = \emptyset \rightarrow (R(x, y) \wedge R(y, z))\}$, which makes that $q$ does not return a function (and thus the modification undefined) whenever $q_{\text{undef}}$ is nonempty. Note furthermore that the update is also undefined if $q^{(2)}_{\text{undef}} := \{(x, y) \mid R(x, y) \wedge \exists^{\geq 2} z\, R(y, z)\}$ is nonempty (here $\exists^{\geq 2} z$ is an abbreviation for "there exist at least two distinct $z$"). In summary, we can write a "safe" version of the update by using for $q$ the query

$$\{(x, y, x, z) \mid R(x, y) \wedge (q^{(2)}_{\text{undef}}(x, y) \rightarrow z = y) \wedge (\neg q^{(2)}_{\text{undef}}(x, y) \rightarrow R(y, z))\}.$$

Since this query returns a function on all instances, the corresponding modification update is always defined. Tuples in $q_{\text{undef}} \cup q^{(2)}_{\text{undef}}$ are left untouched.

*Remark* 3.3. One may wonder how it can be guaranteed that a $2k$-ary query $q$ returns a function on all inputs, or on all inputs that satisfy a given set $\Sigma$ of integrity constraints. For a $2k$-ary relation $r$, requiring that $r$ is a function on $k$-tuples amounts to the constraint on $r$ that the first $k$ columns form a superkey, which is a special kind of functional dependency (FD). Given a conjunctive query $q$, a set $\Sigma$ of FDs, and an FD $\sigma$, the implied constraint problem that $q(I)$ satisfies $\sigma$ for every $I$ that satisfies $\Sigma$, can be solved by the chase algorithm [2, 24]. For first-order queries, the property of always returning a function is undecidable. It is an interesting research topic to see if there exists a syntactic fragment of the first-order queries that would be expressively complete for the first-order function-returning queries in the presence of FDs. □

That queries must return functions poses a serious limitation on the expressive power of modifications. The following technical lemma shows that if there are too many symmetries in the instance, it is very difficult for a query to return a function. Recall that an *automorphism* of an instance $I$ is a permutation $\rho$ such that $\rho(I) = I$. When an automorphism $\rho$ is the identity on $C \subseteq \mathbf{dom}$, then $\rho$ is also called a $C$-automorphism. We say that $\rho$ *fixes* a tuple $t$ simply when $\rho(t) = t$.

**Lemma 3.4.** *Let $\mathbf{S}$ be a database schema, let $q$ be a $C$-generic, $2k$-ary query over $\mathbf{S}$, and let $I$ be an instance of $\mathbf{S}$. If $q(I)$ is a function, then for every pair of $k$-tuples $(t, t') \in q(I)$, the tuple $t'$ is* $C$-fixed *with respect to $(I, t)$, meaning that every $C$-automorphism of $I$ that fixes $t$ also fixes $t'$.*

*Proof.* For the sake of contradiction, suppose $\rho$ is a $C$-automorphism of $I$ that fixes $t$ but not $t'$, so $\rho(t') \neq t'$. We have $\rho(t, t') = (t, \rho(t')) \in \rho(q(I)) = q(\rho(I)) = q(I)$, whence $q(I)$ is not a function, a contradiction. □

*Example* 3.5. Let $\mathbf{S}$ consist of a single unary relation name $S$ and let $I$ be an instance of $\mathbf{S}$ where $I(S) = \{a, b, c, d\}$. Using $a$ and $b$ as constants, we can easily modify $a$ in $S$ to $b$, by using the modification $\mathsf{modify}_S(\{(a, b)\})$. Here,

$\{(a,b)\}$ denotes the constant query $q$ that always outputs $\{(a,b)\}$; note that this query is $\{a,b\}$-generic. In SQL, we would write (using attribute name $A$ for the single column of $S$) the statement update S set A=b where A=a.

But no modification $\mathsf{modify}_S(q)$, with $q$ any $\{a,b\}$-generic query, is able to modify $a$ to $c$. Indeed, $c$ is not $\{a,b\}$-fixed with respect to $(I,a)$, as witnessed by the permutation that swaps $c$ and $d$ but leaves $a$ and $b$ fixed. Hence, by the above Lemma, $(a,c)$ cannot be in $q(I)$.

### 3.3 The update languages $\mathcal{UL}$ and $\mathcal{UL}^{\text{while}}$

Let $\mathbf{S}$ be a database schema. We define the *update programs* over $\mathbf{S}$ as follows.

- If $R \in \mathbf{S}$ is a relation name of arity $k$ and $q$ is a $k$-ary query over $\mathbf{S}$, then '$\mathsf{insert}_R(q)$' and '$\mathsf{delete}_R(q)$' are programs.
- If $R \in \mathbf{S}$ is a relation name of arity $k$ and $q$ is a $2k$-ary query over $\mathbf{S}$, then '$\mathsf{modify}_R(q)$' is a program.
- If $P_1$ and $P_2$ are programs, then $P_1; P_2$ is also a program.
- If $q$ is a boolean query over $\mathbf{S}$ and $P_1$ and $P_2$ are programs, then 'if $q$ then $P_1$ else $P_2$ endif' is a program.
- If $q$ is a boolean query over $\mathbf{S}$ and $P$ is a program, then 'while $q$ do $P$ enddo' is a program.

The language $\mathcal{UL}$ is formed by all programs that do not use while-loops; the extended language allowing while-loops is denoted by $\mathcal{UL}^{\text{while}}$.

Every program denotes an update in the obvious manner. The construct $P_1; P_2$ signifies sequential composition, and if-then-else statements and while-loops have the familiar meaning. A program that contains modifications may not be well defined on every input. Indeed we agree that when a program running on an instance $I$ encounters a modification step that is not defined on the current instance, the entire program is undefined on $I$. Furthermore, programs may be undefined on some instances due to nonterminating while-loops.

Note that every program $P$ is indeed $C$-generic for some $C$. Specifically, let $\mathcal{Q}$ be the set of all queries used in $P$. This set is finite. Each $q$ in $\mathcal{Q}$ is $C_q$-generic for some finite set $C_q$. Then take the union $C = \bigcup_{q \in \mathcal{Q}} C_q$.

**FO-programs** Most of our inexpressibility results concern arbitrary programs, which may use arbitrary queries inside the update operations. But when writing down a program, we need a query language to express the queries. In all our positive results (results where we have to give a specific program) we will be sufficient with first-order logic (FO) as the query language; programs using first-order queries are called *FO-programs.* In writing FO-programs we will often use a mix of relational algebra and relational calculus whenever convenient.

*Example* 3.6. A simple example of an FO-program over a database schema with relations $R$, $S$ and $T$, where $R$ and $T$ have the same arity, is the following:

$$\text{if } S \neq \emptyset \text{ then } \mathsf{insert}_T(R) \text{ else } \mathsf{delete}_T(R) \text{ endif.}$$

This program inserts all of $R$ in $T$ if $S$ is nonempty, and deletes all of $R$ from $T$ otherwise. Note that this program can be equivalently written without an if-then-else construct as follows:

$\mathsf{insert}_T(\{x \mid R(x) \wedge S \neq \emptyset\});$
$\mathsf{delete}_T(\{x \mid R(x) \wedge S = \emptyset\}).$

Clearly this works because conditionals can be expressed in queries, but also because this example program has a simple behavior. In particular, the first statement does not change the relation $S$, so that the condition in the second statement is not influenced by the execution of the first statement. We will see later that, in general, if-then-else constructs cannot be eliminated from the language.

For an example of a program in $\mathcal{UL}^{\text{while}}$, the following FO-program closes off binary relation $R$ transitively:

while $\exists x,y,z(R(x,y) \wedge R(y,z) \wedge \neg R(x,z))$ do
  $\mathsf{insert}_R(\{(x,z) \mid \exists y(R(x,y) \wedge R(y,z))\})$
enddo

# 4 Expressiveness of update primitives

The update language defined in the previous section features the three ubiquitous update primitives, which can be combined using sequential composition, if-then-else, and while-loops. Two natural questions arise as to the expressive power of this language.

The first question concerns replacement. Earlier we have defined replacement as the most general update primitive, which simply replaces a relation with a prescribed new content, given by a query. Is every replacement operation expressible by a program in our language?

The second question concerns the minimality of the language. Is every feature of the language really primitive? It is quite obvious that deletions cannot be eliminated, as they are the only construct that allow a relation to be erased. Likewise it is obvious that insertions cannot be eliminated, as they are the only construct that allow the cardinality of a relation to increase. Also while-loops can clearly not be eliminated. Hence the question of minimality will be focused on modifications and the if-then-else construct.

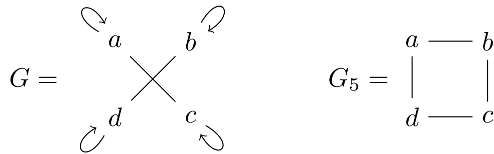
## 4.1 Inexpressibility of replacement

Recall that the *active domain* of an instance $I$, denoted by $adom(I)$, is the set of all data elements appearing in the relations of $I$. Now fix the database schema $\mathbf{S}_{\text{graph}}$ consisting of a single relation name $R$ of arity 2. Each instance $I$ of $\mathbf{S}_{\text{graph}}$ can be viewed as a directed graph $(V, E)$ where $V = adom(I)$ and $E = I(R)$. Consider the *complementation* query, denoted by $R^c$, over $\mathbf{S}_{\text{graph}}$, defined by $R^c(I) = adom(I)^2 - I(R)$

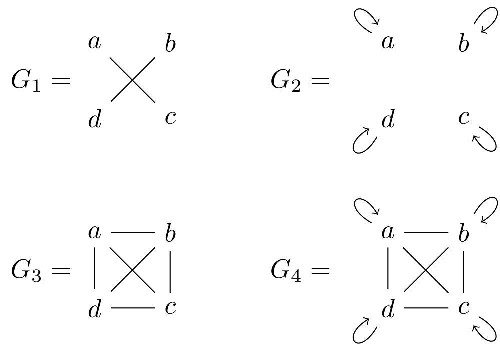
We now show that complementing the set of edges of a directed graph is not possible by any update program.

**Theorem 4.1.** *No update program in* $\mathcal{UL}^{\text{while}}$ *over* $\mathbf{S}_{\text{graph}}$ *can express the update* $\mathsf{replace}_R(R^c)$

*Proof.* Let us denote $\mathsf{replace}_R(R^c)$ by $u$. Consider an instance $I$ where $I(R)$ is of the form $\{(a,c),(c,a),(b,d),(d,b),(a,a),(b,b),(c,c),(d,d)\}$, for four distinct data elements $a$, $b$, $c$ and $d$. We can view $I$ as a graph $G$ with undirected (symmetric) edges and a loop at each node. Similarly, $u(I)$ then equals the graph $G_5$:




We also consider the following graphs:




Finally the empty instance over $\mathbf{S}_{\text{graph}}$ will be denoted by $G_0$.

We also need a notion of "adornment", defined as follows. Let $C$ be a finite set of data elements, disjoint from $\{a,b,c,d\}$. For any fixed $z \in C$, we call the sets $\{(a,z),(b,z),(c,z),(d,z)\}$ and $\{(z,a),(z,b),(z,c),(z,d)\}$ *$C$-adornments*. Edges belonging to such adornments are called *adornment edges*. Also any subset of $C \times C$ is called a $C$-adornment. Edges belonging to such adornments are called *constant edges*. Finally a union of $C$-adornments is also called a $C$-adornment. Now for any graph $G'$ on the nodes $\{a,b,c,d\}$ and any $C$-adornment $Z$, we call $G \cup Z$ a *$C$-adorned version* of $G'$.

We can now make the following claim: *Let $P$ be a sequence of $C$-generic insertions, deletions, and modifications, where $C$ is disjoint from $\{a,b,c,d\}$. Let $G'$ be a $C$-adorned version of $G$ or $G_1$. Then $P(G')$ equals a $C$-adorned version of $G$ or of one of $G_0$–$G_4$.* Assuming this claim, the theorem follows readily. Indeed, take any program $P$. We know that $P$ is $C$-generic for some $C$. Choose $a$, $b$, $c$ and $d$ not in $C$. The execution of $P$ on $I$ traces out a sequence of $C$-generic insertions, deletions, and modifications. Hence, by the claim, the result of $P$ on $I$ is a $C$-adorned version of $G$ or of one of $G_0$–$G_4$, and not the desired result $G_5$.

We prove the claim by induction on the length of the sequence $P$. For the empty sequence the claim holds trivially. Now consider a sequence $u; P$; there are three possible cases for the first update $u$.

The first case is that $u$ is $\mathsf{modify}_R(q)$ for some $C$-generic, 4-ary query $q$. We will take $G'$ to be a $C$-adorned version of $G$; the argument for $G_1$ is similar but simpler. By $C$-genericity of $q$, we know that every $C$-automorphism of $G'$ is also an automorphism of $q(G')$. When we make use of this fact below, we will simply write "by symmetry". Note that the $C$-automorphisms of $G'$ are the symmetries of the square, which form the dihedral group of order 4. Specifically, they are the eight permutations of $\{a,b,c,d\}$ generated by the rotation $(a\ b\ c\ d)$ and the reflection $(a\ b)(c\ d)$.

Consider the possibilities for the modification made by $u$ to edge $(a,c)$ in $G'$. By Lemma 3.4, the new tuple must be fixed by every $C$-automorphism of $G$ that fixes $(a,c)$. Thus there are four possibilities:[4]

1. $(a,c,a,c) \in q(G)$ or $(a,c,c,a) \in q(G)$, i.e., $u$ modifies $(a,c)$ to itself, or to its reversal.
2. $(a,c,a,a) \in q(G)$ or $(a,c,c,c) \in q(G)$, i.e., $u$ modifies $(a,c)$ to a loop at its tail, or at its head.[5]
3. $(a,c,a,x) \in q(G)$, or $(a,c,c,x) \in q(G)$, or $(a,c,x,a) \in q(G)$, or $(a,c,x,c) \in q(G)$ for some $x \in C$, i.e., i.e., $u$ modifies $(a,c)$ to an adornment edge between its head or tail and a constant in $C$.
4. $(a,c,x,x') \in q(G)$ for some $x,x' \in C$, i.e., $u$ modifies $(a,c)$ to a constant edge.

By symmetry, $u$ has the same behavior on the three other edges $(c,a)$, $(b,d)$ and $(d,b)$ as it has on $(a,c)$. So, exactly one of the above four cases applies for all these four edges uniformly.

Always using Lemma 3.4, we can similarly list the possibilities for the modification made by $u$ to the loops in $G'$, and to the adornment edges. Constant edges, by symmetry, can only be modified to constant edges. By going

[4] There is also the possibility that $(a,c)$ is not present in $\pi_{1,2}(q(G))$, but this possibility has the same consequences as possibility 1.

[5] For a directed edge $e = (x,y)$, the node $x$ is called the *tail* of $e$ and $y$ is called the *head* [9].

through all combinations of possibilities one then verifies that $u(G')$ must be of one of the desired forms. The proof is continued in the Appendix. □

## 4.2 Primitivity of modify

The five constructs of the language $\mathcal{UL}^{\text{while}}$ are insertion, deletion, modification, if-then-else, and while-loops. In general, we say that a construct is *redundant* if for every database schema $\mathbf{S}$ and every program $P$ over $\mathbf{S}$ that uses the construct, $P$ is equivalent to a program over $\mathbf{S}$ that does not use the construct. We say that a construct is *primitive* if it is not redundant. We now show that modifications are primitive in this sense. Consider again the database schema $\mathbf{S}_{\text{graph}}$ of directed graphs and the FO query $q_{\text{rev}} \equiv \{(x, y, y, x) \mid R(x, y)\}$ over $\mathbf{S}_{\text{graph}}$. Note that $\mathsf{modify}_R(q_{\text{rev}})$ is the update that reverses all edges of the graph.

**Theorem 4.2.** *No update program in $\mathcal{UL}^{\text{while}}$ over $\mathbf{S}_{\text{graph}}$, that does not use modifications, can express the update $\mathsf{modify}_R(q_{\text{rev}})$.*

The theorem is proved like Theorem 4.1 by a nonreachability argument, based on symmetry and Lemma 3.4. The proof is described in the appendix.

## 4.3 Primitivity of if-then-else

That if-then-else is primitive is not entirely trivial, because queries used in update operations can be arbitrary powerful and express conditionals. This was already illustrated in Examples 3.2 and 3.6. We will show that if-then-else is primitive in $\mathcal{UL}$ and in all its fragments, with the caveat that some of our results assume programs without constants.

Formally, for a nonempty subset $\mathcal{F}$ of $\{\mathsf{insert}, \mathsf{delete}, \mathsf{modify}\}$, we define the fragment $\mathcal{UL}(\mathcal{F})$ consisting of all $\mathcal{UL}$ programs that can use if-then-else in addition to only the update primitives in $\mathcal{F}$. A program not using if-then-else is called a *straight-line* program. It will be convenient to also allow the empty straight-line program and agree it expresses the identity update.

### 4.3.1 Inexpressibility by straight-line programs

The following useful lemma, which may be interesting in its own right, brings out a limitation of straight-line programs.

**Lemma 4.3.** *If a straight-line program $P$ expresses a total and injective update, then $P$ is equivalent to the subsequence of $P$ consisting of all the modifications of $P$.*

The proof of this Lemma, which is given in the Appendix, is based on another lemma:

**Lemma 4.4.** *Every total, injective update is also surjective.*

By Lemma 4.3, in order to find an update not expressible by any straight-line program, it suffices to find an update that is total, injective, and not expressible by modifications only. Over the by now familiar database schema $\mathbf{S}_{\text{graph}}$, we can define such an update, which, moreover, is expressible by an FO-program in $\mathcal{UL}(\{\mathsf{insert}, \mathsf{delete}\})$ as well as in $\mathcal{UL}(\{\mathsf{insert}, \mathsf{modify}\})$. The details are given in the Appendix, allowing us to conclude:

**Theorem 4.5.** *There exists an update over $\mathbf{S}_{\text{graph}}$ that is expressible both by an FO-program in $\mathcal{UL}(\{\mathsf{insert}, \mathsf{delete}\})$ and by an FO-program in $\mathcal{UL}(\{\mathsf{insert}, \mathsf{modify}\})$, but not expressible by any straight-line program.*

It immediately follows that if-then-else is primitive in the two mentioned fragments, as well as in the full language $\mathcal{UL}$.

### 4.3.2 The fragment $\{\mathsf{delete}, \mathsf{modify}\}$

For the fragment $\{\mathsf{delete}, \mathsf{modify}\}$ a result as sharp as Theorem 4.5 remains open. We still can show, however, that if-then-else is primitive in this fragment, except that we have not proven this for programs with constants. Programs with constants can make various markings in a graph, so that it becomes trickier to prove that they are unable to achieve a certain task.

Formally, a program *without constants* is a program in which all queries used in update operations are $C$-generic with $C = \emptyset$. We say that if-then-else is *primitive without constants* if there exists a program without constants but using if-then-else, that is not equivalent to a straight-line program without constants.

**Proposition 4.6.** *Over the schema $\mathbf{S}_{\text{graph}}$, if-then-else is primitive without constants in the fragment $\mathcal{UL}(\mathsf{delete}, \mathsf{modify})$.*

*Proof.* Let $u$ be the update over $\mathbf{S}_{\text{graph}}$ expressed by the following FO-program:

if $\exists x\, R(x, x)$ then
  $\mathsf{delete}_R(\{(x, x) \mid R(x, x)\})$;
  $\mathsf{modify}_R(\{(x, y, x, x) \mid R(x, y)\})$
else *(do nothing)* endif

Here, *(do nothing)* can be expressed by, say, $\mathsf{delete}_R(\emptyset)$.

Suppose, for the sake of contradiction, that $u$ is expressed by an $\emptyset$-generic straight-line program $P$ in $\mathcal{UL}(\mathsf{delete}, \mathsf{modify})$. Choose nine data elements $a$, $b$, …, $h$, $i$ and consider the following graph:

$$G = \begin{array}{cccccc} & & & g & h & i \\ & & & \uparrow & \uparrow & \uparrow \\ \circlearrowleft & \circlearrowleft & \circlearrowleft & & & \\ a & b & c & d & e & f \end{array}$$

We have

$$u(G) = \begin{array}{ccc} \circlearrowleft & \circlearrowleft & \circlearrowleft \\ d & e & f \end{array}$$

We cannot use an unreachability argument as in the proofs of Theorems 4.1 and 4.2, as indeed, $u(G)$ is reachable from $G$ by deletions and modifications. Instead, we use a fooling argument. The idea is that at some point in the execution of $P$, the program is confused whether it is working on input $G$ or on the subset of $G$ without the loops. The remainder of the proof is given in the Appendix. □

#### 4.3.3 The single-primitive fragments

What about the fragments $\mathcal{UL}(\mathsf{insert})$, $\mathcal{UL}(\mathsf{delete})$, and $\mathcal{UL}(\mathsf{modify})$? We can show that if-then-else is still primitive in these fragments, but no longer over the schema $\mathbf{S}_{\text{graph}}$, by the following easy proposition:

**Proposition 4.7.** *Over any database schema that consists of a single relation name, if-then-else is redundant in the fragments* $\mathcal{UL}(\mathsf{insert})$*,* $\mathcal{UL}(\mathsf{delete})$*, and* $\mathcal{UL}(\mathsf{modify})$*.*

The proof of this proposition, given in the Appendix, is based on the following lemma:

**Lemma 4.8.** *Let* $\mathbf{S}$ *be a single-relation database schema and let* $m \in \{\mathsf{insert}, \mathsf{delete}, \mathsf{modify}\}$*. Then every straight-line program* $P$ *over* $\mathbf{S}$ *that uses only operations of kind* $m$*, is equivalent to a single operation of kind* $m$*.*

So we must go to multiple-relation database schemas. It turns out that unary relations are already sufficient now. For the fragment $\mathcal{UL}(\mathsf{modify})$, our result is again without constants only. The proof is given in the Appendix.

**Proposition 4.9.** *Let* $R$*,* $S$ *and* $T$ *be unary relation names.*

- *Over the schema* $\{R, S\}$*, if-then-else is primitive in* $\mathcal{UL}(\mathsf{insert})$*.*
- *Over the schema* $\{R, S, T\}$*, if-then-else is primitive in* $\mathcal{UL}(\mathsf{delete})$*.*
- *Over the schema* $\{R, S, T\}$*, if-then-else is primitive without constants in* $\mathcal{UL}(\mathsf{modify})$*.*

*Remark* 4.10. Proving the above proposition requires some creativity in coming up with the inexpressible updates, but is otherwise quite straightforward, because each update operation can act on a single relation only. It may be interesting to consider update operations that can change multiple relations in parallel. Such operations are not provided in SQL.

# 5 Alternation

In this section we present some initial results on the expressive power of alternating different update primitives. Everything else in this direction remains to be further explored. We work over the database schema $\mathbf{S}_{\text{graph}}$ of directed graphs; an analogous investigation could be performed over other schemas as well. Another caveat is that we restrict attention to updates without constants (formally, $C$-generic updates with $C = \emptyset$).

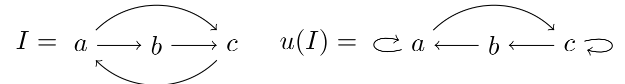
For $m', m' \in \{\mathsf{insert}, \mathsf{delete}, \mathsf{modify}\}$, an *update of the form* $m; m'$ is a composition of two update operations $\text{op}_1; \text{op}_2$ where $\text{op}_1$ is of kind $m$ and $\text{op}_2$ is of kind $m'$.

**Theorem 5.1.** *Let* $m, m' \in \{\mathsf{insert}, \mathsf{delete}, \mathsf{modify}\}$ *be two different update primitives. Then there exists a first-order update over* $\mathbf{S}_{\text{graph}}$*, without constants, of the form* $m; m'$*, that is not equivalent to any update, without constants, of the form* $m'; m$*.*

The lengthy proof is given in the Appendix and gives in each of the six cases an inexpressible update. The essential idea is always that information loss cannot be prevented when we have to start with an unsuitable operation. More specifically, for each update, we choose an input with some symmetries. Then after the first step, these symmetries cause generic queries to be confused how to proceed on an intermediate result that may correspond to different inputs with different outputs.

The above theorem shows that between any two primitives, the two different forms of one alternation (starting with either of the two primitives) can be separated. Much more generally one would expect a hierarchy in analogy (but not more than an analogy) to the quantifier alternation hierarchy for first-order queries on relational databases [15]. The question remains to be explored. We can only offer an example in the fragment $\mathcal{UL}(\mathsf{insert}, \mathsf{delete})$ of a straight-line program with two alternations that cannot be expressed using only one alternation.

*Example* 5.2. Over the schema $\mathbf{S}_{\text{graph}}$ consider the update $u \equiv \mathsf{replace}_R(R \circ R)$, where $R \circ R \equiv \{(x, z) \mid \exists y(R(x, y) \wedge R(y, z))\}$. We conjecture that this update is not expressible in $\mathcal{UL}^{\text{while}}$ at all. At least we can show that $u$ is not expressible by any straight-line program in $\mathcal{UL}(\mathsf{insert}, \mathsf{delete})$ that is without constants and uses only one alternation. Since $\mathbf{S}_{\text{graph}}$ has only a single relation name, by Lemma 4.8, such a program is of the form $\mathsf{insert}; \mathsf{delete}$ or $\mathsf{delete}; \mathsf{insert}$. Consider the following instance $I$ and its updated version $u(I)$:




First, consider any program $P$, without constants, of the form $\mathsf{insert}; \mathsf{delete}$. For $P$ to compute $u(I)$ on input $I$, the insertion step must insert in $u(I)$ that are not in $I$, yielding an instance that has the transposition $(a\ c)$ as an automorphism. Since $u(I)$ does not have this transposition as an automorphism, we cannot go from the intermediate result

to $u(I)$ by an $\emptyset$-generic update. By a similar argument we can see that no program of the form delete; insert can compute $u(I)$ on input $I$.

Nevertheless, the following ad-hoc program does compute $u(I)$ from $I$ using two alternations: delete $(c, a)$; insert $R \circ R$; delete $(a, b)$ and $(b, c)$. Here, in this ad-hoc program, $a$, $b$ and $c$ are *not* used as constants but can be distinguished on the relevant intermediate results by generic queries.

# 6 Expressing replace

As already mentioned in the Introduction, the limitations in expressive power of the language $\mathcal{UL}^{\text{while}}$ as illustrated by Theorems 4.1 and 4.2 vanish in the presence of temporary relations. Temporary relations can be elegantly formalized as follows [5]. Let $u$ be an update over some database schema $\mathbf{S}$. To express $u$ by a program $P$, we allow $P$ to be a program over a larger schema $\mathbf{S}' \supseteq \mathbf{S}$. When given an instance of $\mathbf{S}$ as input to $P$, the relations outside $\mathbf{S}$ are initialized to the empty set. The final result of $P$, an instance of $\mathbf{S}'$, is restricted to the relations from $\mathbf{S}$. In the presence of temporary relations, there is not that much difference anymore between an update language and a general query language. At any rate, the expressive power of $\mathcal{UL}^{\text{while}}$ with temporary relations is quite large and well-understood. When the queries used inside programs are first-order, the expressible queries (or updates) are known as the *while*-queries, or the queries expressible in FO(PFP), the extension of first-order logic with partial fixpoints [2].

In SQL practice, however, the use of temporary relations can be cumbersome, perhaps harder to optimize, and in-place update operations seem to be preferred by SQL programmers. To wit, the SQL community has standardized two extensions of the basic insert–delete–modify update repertoire in the recent SQL:2011 standard: the merge statement and *data change delta tables* [29]. We will show that these two extensions both allow to express the general replace primitive.

## 6.1 Merge

The SQL:2011 merge statement is a quite complex instruction that involves a combination of insertions, deletions, and modifications, performed on a target relation by processing a source relation [33]. We refer to the DB2 documentation for a detailed description [16]. We offer the following formalization.

Let $R$ be a relation name of arity $k$ and let $q_{\text{source}}$ be a query of arity $l$. Furthermore, let $q_{\text{match}}$ be a query of arity $k + l$; let $q_{\text{update}}$ be a query of arity $2k + l$; and let $q_{\text{delete}}$ and $q_{\text{insert}}$ be queries of arity $k + l$. Then $\mathsf{merge}_R(q_{\text{source}}, q_{\text{match}}, q_{\text{update}}, q_{\text{delete}}, q_{\text{insert}})$ is the update $u$ defined as follows.

Let $I$ be an instance. We define the following relations:

- $r_{\text{source}} = q_{\text{source}}(I)$. This formalizes the source relation of the SQL merge statement.
- $r_{\text{match}} = \{(t, s) \in q_{\text{match}}(I) \mid t \in I(R) \wedge s \in r_{\text{source}}\}$. This formalizes that target tuple $t$ and source tuple $s$ match.
- $r_{\text{update}} = \{(t, s, t') \in q_{\text{update}}(I) \mid (t, s) \in r_{\text{match}}\}$. This represents that the matching pair $(t, s)$ qualifies the conditions to do an update ('update' in the SQL sense); $t'$ is the modified tuple.
- $r_{\text{delete}} = q_{\text{delete}}(I) \cap r_{\text{match}}$. This returns the matching pairs that qualify to do a delete.
- $r_{\text{nomatch}} = r_{\text{source}} - \pi_{k+1,\dots,k+l}(r_{\text{match}})$. This returns the source tuples that do not match.
- $r_{\text{insert}} = \{(s, t) \in q_{\text{insert}}(I) \mid s \in r_{\text{nomatch}}\}$. This represents the insert action specified for source tuples that do not match any target tuple.

For $u$ to be defined on $I$, a number of requirements must be satisfied:

- The first $k$ columns should be a superkey for $r_{\text{match}}$, i.e., each target tuple can match with at most one source tuple.
- The first $k + l$ columns should be a key for $r_{\text{update}}$, similar to the well-definedness requirement for modifications;
- Relation $r_{\text{delete}}$ must be disjoint from $\pi_{1,\dots,k+l}$ $(r_{\text{update}})$;
- The first $l$ columns should be a key for $r_{\text{insert}}$, i.e., each nonmatched source tuple can insert at most one tuple.

If these requirements are satisfied by $I$, then $u(I)$ replaces the contents of relation $R$ by the relation

$$\big(I(R) - (\pi_{1,\dots,k}(r_{\text{update}}) \cup \pi_{1,\dots,k}(r_{\text{delete}}))\big) \cup \pi_{k+l+1,\dots,2k+l}(r_{\text{update}}) \cup \pi_{l+1,\dots,l+k}(r_{\text{insert}})$$

It is now clear that an arbitrary replacement $\mathsf{replace}_R(q)$ can be expressed by merge using the following queries: $q_{\text{source}}$ is $(R - q) \cup (q - R)$; $q_{\text{update}}$ is $\emptyset$; and $q_{\text{match}}$, $q_{\text{delete}}$, and $q_{\text{insert}}$ are the equality query $\{(x_1, \dots, x_k, y_1, \dots, y_k) \mid x_1 = y_1 \wedge \dots \wedge x_k = y_k\}$. Quite simply, all tuples in $R - q$ match for equality and are deleted; all tuples in $q - R$ do not match for equality and are inserted. So, we need only a very simple application of the merge statement. Yet, the crucial feature that was added to SQL:2011 in comparison to earlier standards is that deletions as well as insertions can be used in one merge statement, and it is exactly that feature that allows our replacement procedure to work.

*Example* 6.1. In SQL syntax, over a binary relation R(A,B), the following statement replaces $R$ by $R \circ R$ (compare Example 5.2):

```
merge into R
using (
  ((select A,B from R)
   except
   (select R1.A,R2.B from R R1, R R2
    where R1.B=R2.A))
  union
  ((select R1.A,R2.B from R R1, R R2
    where R1.B=R2.A)
   except
   (select A,B from R)))
  as S(A,B)
on R.A=S.A and R.B=S.B
when matched then delete
when not matched then insert values (S.A,S.B)
```

### 6.2 Data change delta tables

Data change delta tables are a feature of SQL:2011 that allow update operations to be put inside queries. The table before the update, as well as the table after the update, can be accessed by the query. Data change delta tables can be used to perform arbitrary in-place replacement updates, when used in conjunction with the with clause of select statements. The with clause allows intermediate queries to be given a temporary name inside a larger query, and is better known as the way to specify recursion in SQL; here we do not use recursion.

Specifically, remember the procedure to perform a replacement $\mathsf{replace}_R(q)$ using a scratch relation $S$: insert $q$ into $S$; erase $R$; and insert $S$ into $R$. This procedure can be almost literally programmed as follows:

```
with S as (q),
     Dummy as (select * from old table (delete from R))
select *
from new table (insert into R (select * from S))
```

The specifications old table and new table are not actually important for the above to work. So, we need only a very limited application of data change delta tables; the only feature we really need is the ability to put updates in queries, and the ability to simulate temporary relations using the with clause.

### 6.3 Value invention

The classical definitions of $C$-generic query and update imply that the active domain of the output $q(I)$ is a subset of the active domain of the input $I$ (plus the constants in $C$). One can extend the notion of generic query and update, however, to allow for *value invention*: the introduction of new data elements in the result [5, 3, 31, 2].

We now describe an, admittedly artificial, technique to perform $\mathsf{replace}_R(q)$, with $q$ a classical $C$-generic query, using only in-place insertions and deletions on $R$, if the queries allowed inside insertions can do value invention. So, value invention is used as an auxiliary mechanism only. Moreover, the query language must be sufficiently powerful to do counting and iteration. Although the combination of iteration and value invention in general leads to Turing completeness, our technique has only polynomial complexity. It remains open if replacement can still be simulated if only first-order logic, extended with value invention, is permitted inside the queries.

Concretely, we assume $R$ to be binary, but the method can be adapted to higher arities. We proceed in four steps.

**Encoding:** Let $n$ be the cardinality of $adom(R) \cup C$. Then for every edge $e = (a, b) \in R$ we insert in $R$ a chain of $n + 2$ new data elements $(e_0, e_1)$, $(e_1, e_2)$, $\dots$, $(e_n, e_{n+1})$, along with edges $(e_n, b)$ and $(e_0, a)$ and the loop $(e_{n+1}, e_{n+1})$. Note that after this insertion, the newly introduced data elements are indistinguishable from the original data elements except by their structural properties. And indeed, they can still be structurally distinguished as follows. In the instance after the insertion, call a *sl-chain* any chain of distinct data elements starting in a source node (node without entering edges) and ending in a loop. The *length* of a chain is the number of elements on it. Let $m$ be the maximum length of an sl-chain. Chains of distinct nodes in the original graph can be at most $n$ long. The start node of such a chain is linked from its new $e_0$ element, making a total length of $n+1$. Since the new elements form sl-chains of length $n+2$, they can be distinguished by their lying on an sl-chain of maximum length $m$.

**Remove original edges:** After the encoding insertion step, the original graph is still definable. Indeed, it consists of all pairs $(a, b)$ such that there is an sl-chain $e_0 \dots e_m$ and edges $(e_0, a)$ and $(e_m, b)$. Note that $a$ can be distinguished from $e_1$ by the maximum length of the chain as explained above. Hence, we can delete the original edges without loss of information.

**Insert $q$:** We can now determine the result of query $q$ on the original instance, since the original relation $R$ is still encoded in the current relation $R$. The result is inserted into $R$. Note that, since $q$ is a classical $C$-generic query, this inserts only edges between original elements of $R$. In particular, as before, this insertion cannot introduce sl-chains of length longer than $n+1$.

**Remove encoding:** We can finally remove all elements lying on a maximal-length sl-chain, and we are left with the desired value of the replacement.

### 6.4 Arithmetic

Also when interpreting the data elements in relations as numbers on which arithmetic can be performed, we leave the framework of generic queries. Queries again no longer need to be domain-preserving, as witnessed by the simple SQL query select A+B from R.

In this context we can simulate replacement in a way similar to the simulation using value invention, but simpler, since no iteration over long chains is needed anymore. In fact, the whole procedure can now be programmed in SQL as a sequence of insert and delete statements using normal query expressions, i.e., without using the programming facilities of SQL/PSM. The arithmetical operators needed below are order comparisons, addition, and the aggregate functions max and count. It would be interesting to understand better exactly how much (or how little) arithmetic is really needed.

The encoding step is now much simpler. Let $M$ be the maximum number appearing in both relations $R$ and $q(R)$. Also, for any edge $e = (a, b)$ in $R$, let $n_e$ be its rank in a lexicographic ordering of the tuples of $R$. Then we encode $e$ by inserting the edges $(e_0, e_1)$, $(e_1, b)$, $(e_0, a)$, where now $e_0$ and $e_1$ are no longer abstract new data elements, but the numbers $e_0 = M + 2(n_e - 1) + 1$ and $e_1 = e_0 + 1$.

After inserting these edge encodings, the original edges can still be distinguished. The source nodes are exactly all elements $e_0$. The two edges leaving a source node $e_0$ are $(e_0, a)$ and $(e_0, e_1)$, and $a$ can be distinguished from $e_1$ simply by $a < e_1$. Then $b$ can be retrieved as the only node pointed to by $e_1$.

# 7 Conclusion

Theoretical computer science has a rich tradition of investigating its computational models to the bone, with the goal of understanding the power and complexity of each individual feature. In database theory we have certainly followed this tradition in the investigation of high-level logical query languages. Computational models for updating deserve the same attention, because of their practical interest as witnessed by additions to recent SQL standards.

In this paper we have scratched the surface of much that remains to be explored. Throughout the text we have identified open problems, mostly of a technical or theoretical nature. One theory-oriented question we have not yet mentioned is the arity required of temporary relations, similarly to questions investigated about the arity required of auxiliary relations in first-order incremental evaluation systems (see the recent paper [20] and references therein).

Here we conclude with some directions for further research. In Section 2 we have already mentioned the obvious direction of working with other data models than the relational model.

We have focused on updating a single relation, possibly part of a multi-relation database. As such, replacement of a single relation by a query applied to the database was the natural upper bound on expressive power for our investigation. However, in general one wants to perform multi-relation updates and much of our study needs to be reconsidered for that context.

In this paper we have also focused on sequential composition as a natural way of executing programs. Database servers in practice also process sequences of SQL statements. Yet it seems interesting to also study parallel composition of updates. Also, in practice, updates often happen through cursors. It would be interesting to model this by a formally defined programming language, so that again the possibilities and limitations of cursor-based SQL programming can be investigated on the theoretical level.

Finally, we have not investigated the efficiency, performance, and optimization aspects of updating. It would be interesting to compare the efficiency of using temporary relations versus other ways of performing complicated updates. Also, it would be interesting to rigorously test the thesis that in-place updates are more efficient than general replacement updates.

# References


[1] S. Abiteboul. Updates, a new frontier. In M. Gyssens, J. Paredaens, and D. Van Gucht, editors, *ICDT'88*, volume 326 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 1988.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. *Journal of the ACM*, 45(5):798–842, 1998.

[4] S. Abiteboul and V. Vianu. Equivalence and optimization of relational transactions. *Journal of the ACM*, 35:70–120, 1988.

[5] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41(2):181–229, 1990.

[6] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[7] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *Conference Record, 6th ACM Symposium on Principles of Programming Languages*, pages 110–120, 1979.

[8] M. Andries, L. Cabibbo, J. Paredaens, and J. Van den Bussche. Applying an update method to a set of receivers. *ACM Transactions on Database Systems*, 25(1):1–40, 2001.

[9] J. Bang-Jensen and G.Z. Gutin. *Digraphs*. Springer, second edition, 2009.

[10] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Verification of tree updates for optimization. In K. Etessami and S.K. Rajamani, editors, *Computer Aided Verification, 17th International Conference*, volume 3576 of *Lecture Notes in Computer Science*, pages 379–393. Springer, 2005.

[11] M. Benedikt and J. Cheney. Semantics, types and effects for XML updates. In Ph. Gardner and F. Geerts, editors, *Proceedings 12th International Symposium on Database Programming Languages*, volume 5708 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.

[12] M. Benedikt and J. Cheney. Destabilizers and independence of XML updates. *Proceedings VLDB*, 3(1):906–917, 2010.

[13] M. Benedikt, D. Florescu, Ph. Gardner, G. Guerrini, M. Mesiti, and E. Waller. Report on the EDBT/ICDT 2010 workshop on updates in XML. *SIGMOD Record*, 39(1):54–57, 2010.

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

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

[16] Database reference–SQL–statements–MERGE. IBM DB2 10.1 Information Center for Linux, UNIX, and Windows, `http://pic.dhe.ibm.com/infocenter/db2luw/v10r1/`, 15 November 2012.

[17] S. Dekeyser, J. Hidders, and J. Paredaens. A transaction model for XML databases. *World Wide Web: Internet and Web Information Systems*, 7:29–57, 2004.

[18] B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors. *Transactions and Change in Logic Databases*, volume 1472 of *Lecture Notes in Computer Science*. Springer, 1998.

[19] G. Ghelli, K. Høgsbro Rose, and J. Siméon. Commutativity analysis for XML updates. *ACM Transactions on Database Systems*, 33(4):article 29, 2008.

[20] E. Grädel and S. Siebertz. Dynamic definability. In *Proceedings 15th International Conference on Database Theory*, 2012.

[21] J. Hidders, J. Paredaens, and R. Vercammen. On the expressive power of XQuery-based update languages. In *Database and XML Technologies, Proceedings 4th XSym*, volume 4156 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2006.

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

[23] D. Karabeg and V. Vianu. Simplification rules and complete axiomatization for relational update transactions. *ACM Transactions on Database Systems*, 16(3):439–475, 1991.

[24] A. Klug. Calculating constraints on relational expressions. *ACM Transactions on Database Systems*, 5(3):260–290, 1980.

[25] C. Laasch and M.H. Scholl. Deterministic semantics of set-oriented update sequences. In *Proceedings, 9th International Conference on Data Engineering*, pages 4–13. IEEE Computer Society Press, 1993.

[26] A.Y. Levy and Y. Sagiv. Queries independent of updates. In R. Agrawal, S. Baker, and D.A. Bell, editors, *Proceedings 19th International Conference on Very Large Data Bases*, pages 171–181. Morgan Kaufmann, 1993.

[27] Udo W. Lipeck and B. Thalheim, editors. *Modelling Database Dynamics*. Workshops in Computing. Springer-Verlag, 1992.

[28] MySQL reference manuals. `http://dev.mysql.com/doc/`.

[29] Information technology — database languages — SQL — part 2: Foundations. International Standard ISO/IEC 9075, 2012.

[30] I. Tatarinov, Z.G. Ives, A.Y. Halevy, and D.S. Weld. Updating XML. In *Proceedings 2001 ACM SIGMOD International Conference on Management of Data*, pages 413–424. ACM Press, 2001.

[31] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating database transformation languages. *Journal of the ACM*, 44(2):272–319, 1997.

[32] XQuery update facility 1.0. W3C Recommendation 17 March 2011.

[33] F. Zemke. What's new in SQL:2011. *SIGMOD Record*, 41(1):67–73, 2012.

# A Appendix

*Proof of Theorem 4.1.* (Continued.) Now consider the possibilities for the modification made by $u$ to the loop-edge $(a, a)$ in $G$. By Lemma 3.4, the new tuple must

be fixed by every $C$-automorphism of $G'$ that fixes $(a, a)$. Thus there are four possibilities:[6]

5. $(a, a, a, a) \in q(G)$, i.e., $u$ modifies $(a, a)$ to itself.

6. $(a, a, a, c) \in q(G)$ or $(a, a, c, a) \in q(G)$, i.e., $u$ modifies $(a, a)$ to the edge leaving $a$ or the edge entering $a$.

7. $(a, a, a, z) \in q(G)$ or $(a, a, z, a) \in q(G)$ for some $z \in C$, i.e., $u$ modifies $(a, a)$ to an adornment edge between $a$ and a constant in $C$.

8. $(a, a, z, z') \in q(G)$ for some $z, z' \in C$, i.e., $u$ modifies the $(a, a)$ to a constant edge.

By symmetry, $u$ has the same behavior on the three other loops $(b, b)$, $(c, c)$, $(d, d)$ as it has on $(a, a)$. So, exactly one of the above four cases applies for all these four loops uniformly.

Similarly, adornment edges $(a, x)$ or $(x, a)$, if existing, can be modified to other adornment edges, to the loop $(a, a)$, or to constant edges. Finally, constant edges, by symmetry, can only be modified to constant edges.

By going through all combinations of possibilities one can verify that $u(G')$ must be a $C$-adorned version of $G$, $G_1$, or $G_2$. If $G$ or $G_1$, the claim follows by induction. If $G_2$, we argue as follows. Note that $u; P(G') = P(u(G'))$. The graph $G_2$, and any of its $C$-adorned versions like $u(G')$, is fully symmetric over $\{a, b, c, d\}$: every permutation of $\{a, b, c, d\}$ is an automorphism of $G_2$. Since $P$ is $C$-generic with $C$ disjoint from $\{a, b, c, d\}$, also $P(u(G'))$ must be fully symmetric. Now the only graphs on $\{a, b, c, d\} \cup C$ that are fully symmetric in the above sense are the $C$-adorned versions of $G_0$, $G_2$, $G_3$ and $G_4$, so the claim follows.

The second case is that $u$ is $\mathsf{insert}_R(q)$ for some $C$-generic, binary query $q$. We assume $G'$ is a $C$-adorned version of $G_1$; the argument for $G$ is similar but simpler. We consider three independent possibilities for $u$ on $G'$:

1. If $(a, a) \in u(G')$, then by symmetry, the other loops $(b, b)$, $(c, c)$ and $(d, d)$ are in $u(G')$ as well.

2. If $(a, b) \in u(G')$, then by symmetry, also $(b, c)$, $(c, d)$, $(d, a)$, and all reversals of these edges are in $u(G')$.

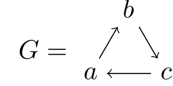
3. If $(a, x) \in u(G')$ or $(x, a) \in u(G')$ for some $x \in C$, then by symmetry, also $(b, x)$, $(c, x)$ and $(d, x)$, or $(x, b)$, $(x, c)$ and $(x, d)$ are in $u(G')$ as well.

Going through all combinations of possibilities we obtain that $u(G')$ must be a $C$-adorned version of $G$, $G_1$, $G_3$, or $G_4$. We can now reason as before. When $G$ or $G_1$, we can apply induction, and when $G_3$ or $G_4$, we are fully symmetric.

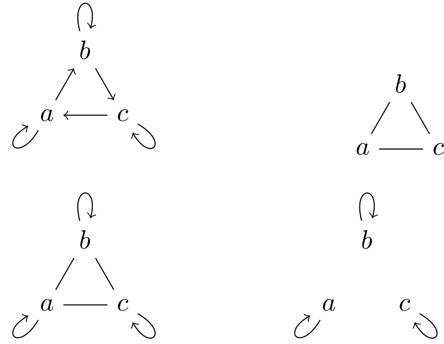
[6]Again there is also the possibility that $(a, a)$ is not present in $\pi_{1,2}(q(G))$, but this possibility has the same consequences as possibility 5.

The final case is that $u$ is $\mathsf{delete}_R(q)$ for some $C$-generic, binary query $q$. Again using symmetry, we can see that $u(G')$ is a $C$-adorned version of $G$, $G_0$, $G_1$, or $G_2$ and we can reason as before. □

*Proof of Theorem 4.2.* (Sketch.) Consider an instance over $\mathbf{S}_{\text{graph}}$ that has the following form viewed as a graph:




We make a similar argument as in the proof of Theorem 4.1, using symmetry. Specifically, one verifies that the only instances reachable from $G$ by $C$-generic insertions and deletions, with $C$ disjoint from $\{a, b, c\}$, are $C$-adorned versions of the empty graph, the graph $G$ itself, and the following four graphs:




Here, lines without arrows denote undirected, i.e., symmetric pairs of, edges. None of these graphs equals the reversal of $G$, and the same reasoning can be applied as in the proof of Theorem 4.1. □

*Proof of Lemma 4.4.* Let $u$ be a total, injective, $C$-generic update over some schema $\mathbf{S}$ and let $J$ be an arbitrary instance of $\mathbf{S}$. We must show that $J = u(I)$ for some instance $I$. Since $u$ is a $C$-generic function, for any instance $I$, we have $\mathit{adom}(u(I)) \subseteq \mathit{adom}(I) \cup C$. Thus consider the set $\mathcal{J} = \{I \in \mathit{inst}(\mathbf{S}) \mid \mathit{adom}(I) \subseteq \mathit{adom}(J) \cup C\}$. The image of $\mathcal{J}$ by $u$ is a subset of $\mathcal{J}$. But then the image of $\mathcal{J}$ by $u$ equals $\mathcal{J}$, since $u$ is total and injective and $\mathcal{J}$ is finite (pigeon-hole principle). Hence, since $J \in \mathcal{J}$, there exists $I \in \mathcal{J}$ such that $u(I) = J$, as desired. □

*Proof of Lemma 4.3.* By induction on the length of $P$. For the empty program, the lemma holds trivially. Now consider a straight-line program of the form op; $P$ where op is an insertion, deletion, or modification, so that the update expressed by op; $P$ is total and injective. In particular, op must already be total and injective in itself.

First, assume op is an insertion. We claim that *any insertion that is not the identity, is not injective.* By this claim, op must be the identity, so op; $P$ is equivalent to $P$ and the lemma follows by induction. To prove the claim,

let us define the partial order $\subseteq$ on instances defined by $I_1 \subseteq I_2$ iff $I_1(R) \subseteq I_2(R)$ for each relation name of the database schema. Since op is an insertion, op is inflationary with respect to this partial order. Since op is not the identity, there is an instance $I$ such that $I \subsetneq \text{op}(I)$. We then consider the increasing sequence $(I_n)_n$ where $I_0 = I$ and $I_{n+1} = \text{op}(I_n)$. Then $adom(I_n) \subseteq adom(I) \cup C$ for every $n$, where $C$ is a finite set such that op is $C$-generic. Since, over a fixed database schema, there are only finitely many possible instances $J$ with $adom(J) \subseteq adom(I) \cup C$, the sequence converges, i.e., there exists $m$ such that $I_m = I_{m+1}$. Choose the least such $m$; note that $m > 0$ because $I_0 \neq I_1$. Then we have $\text{op}(I_{m-1}) = I_m = \text{op}(I_m)$, so op is not injective as desired.

Second, assume op is a deletion. By a similar argument, now using a decreasing sequence, we can show that any deletion that is not the identity is not injective. Then the lemma follows again by induction.

Finally, assume op is a modification. Since op is total and injective, op is also surjective by Lemma 4.4. Since op; $P$ is total and injective and op is surjective, $P$ must be total and injective. By induction, $P$ is equivalent to its sequence of modifications, so the same holds for op; $P$ as desired. □

*Proof of Theorem 4.5.* Consider the following technical condition (∗) on instances $G$ over $\mathbf{S}_{\text{graph}}$. First, there must exist precisely one node, called the "marker", with the property that it has edges to all other nodes; second, the marker must have a loop;[7] and third, every nonmarker node with a loop must have an edge to or from another nonmarker node. Now define the update $u_{\text{CLWM}}$ over $\mathbf{S}_{\text{graph}}$ ("complement loops with marker") as follows. If $G$ does not satisfy condition (∗), then $u_{\text{CLWM}}(G) = G$; else, $u_{\text{CLWM}}(G)$ equals $G$ where a loop is added to all nonmarker nodes that do not have a loop, and the loop is removed from all nonmarker nodes that have a loop.

This update is expressible by an FO-program in $\mathcal{UL}(\{\mathsf{insert}, \mathsf{delete}\})$, as well as in $\mathcal{UL}(\{\mathsf{insert}, \mathsf{modify}\})$. Indeed, condition (∗) is expressible by an FO boolean query. Then the actual update is done in two steps: first, delete all edges from the marker to nonmarker nodes with a loop, and also all loops on these nonmarker nodes; second, insert loops at nodes pointed to by the marker, and also re-insert edges from the marker to all nonmarker nodes. (Note that, after the first step, the marker is still distinguishable as the only node with a loop.) The delete step can be expressed by a modification as well as by a deletion. Indeed, we can delete edges leaving the marker by modifying them to the loop at the marker.

Furthermore, $u_{\text{CLWM}}$ is clearly total, and it can be verified that it is injective. Moreover, $u_{\text{CLWM}}$ is not expressible by modifications only, since modifications cannot increase the cardinality of relations, whereas $u_{\text{CLWM}}$ maps the graph with three edges $\{(a,a),(a,b),(a,c)\}$ (the marker is $a$) to the graph with five edges obtained by adding loops to $b$ and $c$. □

*Proof of Proposition 4.6.* (Continued.) Let us first characterize which graphs can be reached from $G$ by $\emptyset$-generic modifications only. It turns out there are only four such graphs, namely $A \cup B_k$ for $k = 1, \dots, 4$, where $A = \{(a,a),(b,b),(c,c)\}$, $B_1 = \{(d,g),(e,h),(f,i)\}$, $B_2 = \{(g,d),(h,e),(i,f)\}$, $B_3 = \{(d,d),(e,e),(f,f)\}$, and $B_4 = \{(g,g),(h,h),(i,i)\}$. Indeed, using reasoning by symmetry as in the proof of Theorem 4.1, it can be shown that any $\emptyset$-generic modification applied to a graph of the form $A \cup B_k$ yields another graph of the form $A \cup B_l$ with $l \in \{1,2,3,4\}$.

Since $u(G)$ is not of the form $A \cup B_k$, the program $P$ has at least one deletion that is not the identity when executed on input $G$; let $\delta$ be the first such deletion. Thus, consider the prefix $P'$ of $P$ up to, but not including, $\delta$. We know that $G' = P'(G)$ is of the form $A \cup B_k$. We can rule out $B_4$, however, since $A \cup B_4$ has lost the domain elements $d$, $e$ and $f$ which are needed in $u(G)$. We can rule out $B_3$ as well, since then $G'$ would consist of six isolated loops; an $\emptyset$-generic deletion that is not the identity has no choice but entirely erasing an instance with so much symmetry. So, $G'$ equals $A \cup B_1$ or $A \cup B_2$. Now $\delta$ applied to $G'$ cannot delete the edges from $B_k$, as again this would result in the loss of $d$, $e$ and $f$. So, $\delta$ deletes the edges from $A$. (By symmetry, if $\delta$ deletes one edge from $A$ then it deletes all edges from $A$.) We conclude that the prefix $P'; \delta(G)$ equals $B_1$ or $B_2$.

Let us now consider what $P'; \delta$ does on input $B_1$. Reasoning as before, $P'(B_1)$ equals $B_1$ or $B_2$. Then the deletion $\delta$ is not allowed to delete anything. Indeed, by symmetry it would then delete everything, leaving the empty instance, whereas the $u(B_1)$ equals $B_1$ itself. We conclude that $P'; \delta(B_1)$ equals $B_1$ or $B_2$.

Noting that $B_1$ and $B_2$ are isomorphic, we obtain that $P'; \delta(G)$ and $P'; \delta(B_1)$ are isomorphic, whence, by genericity, $P(G)$ and $P(B_1)$ are isomorphic as well. This shows that $P$ is incorrect, since $B_1 = u(B_1)$ is not isomorphic to $u(G)$. □

*Proof of Lemma 4.8.* It suffices to show that the composition of two update operations of kind $m$ is equivalent to a single update operation of kind $m$. For insertions, $\mathsf{insert}_R(q_1); \mathsf{insert}_R(q_2)$ is equivalent to $\mathsf{insert}_R(q_1 \cup q_2(R \cup q_1))$. For deletions, $\mathsf{delete}_R(q_1); \mathsf{delete}_R(q_2)$ is equivalent to $\mathsf{delete}_R(q_1 \cup q_2(R - q_1))$. Finally, consider a composition $\mathsf{modify}_R(q_1); \mathsf{modify}_R(q_2)$. Here, $q_1$ and $q_2$ are $2k$-ary queries with $k$ the arity of $R$. Without loss of generality, we may assume that $q_1$ and $q_2$ return functions (from $k$-tuples to $k$-tuples) that are defined on all tuples of $R$, whenever they return functions at all. Let $q_1^{\text{rep}}$ be the $k$-ary ary such that $\mathsf{modify}_R(q_1)$ is equivalent to $\mathsf{replace}_R(q_1^{\text{rep}})$. Let $q_2'$ be defined by $q_2'(I) = \{(t, t'') \mid \exists t' : (t, t') \in q_1(I)$ and $(t', t'') \in q_2(q_1^{\text{rep}}(I))\}$. Finally let $q_2''$ be defined by

[7] A *loop* in a directed graph is an edge of the form $(x, x)$.

$q''_2(I) = q'_2(I)$ if $q_1(I)$ is a function, and $q''_2(I) = adom(I)^{2k}$ otherwise. Then the composition can be expressed as $\mathsf{modify}_R(q''_2)$. □

*Proof of Proposition 4.7.* Consider a program $P$ of the form if $q$ then $P_1$ else $P_2$ endif. By induction, $P_1$ and $P_2$ may be assumed to have no if-then-else. Consequently, $P_1$ and $P_2$ are compositions of update operations of the same primitive. By Lemma 4.8, $P_1$ and $P_2$ each may be assumed to consist of a single update operation (of the same primitive). It now remains to note that if-then-else can be pushed inside the update operation. Indeed, for any primitive $m \in \{\mathsf{insert}, \mathsf{delete}, \mathsf{modify}\}$, we can see that if $q$ then $m_R(q_1)$ else $m_R(q_2)$ endif is equivalent to $m_R(q')$ where $q'$ is the query if $q$ then $q_1$ else $q_2$ with the obvious semantics. □

*Proof of Proposition 4.9.* For $\mathcal{UL}(\mathsf{insert})$, let $u$ be the update expressed by the following program:

if $R \cap S = \emptyset$ then
  $\mathsf{insert}_R(S)$;
  $\mathsf{insert}_S(R)$
else *(do nothing)* endif

This update cannot be expressed by any straight-line program $P$ using only insertions. In proof, consider the input instance $I = \{(R, \{a\}), (S, \{b\})\}$ with $u(I) = \{(R, \{a, b\}), (S, \{a, b\})\}$. Since each insertion can change one relation only, there is a prefix $P'$ of $P$ such that $P'(I)$ equals either $I_1 = \{(R, \{a, b\}), (S, \{b\})\}$ or $I_2 = \{(R, \{a\}), (S, \{a, b\})\}$. Say it is $I_1$; the argument for $I_2$ is analogous. Since $u(I_1) = I_1$, and we have only insertions, every step of $P$ on $I_1$ should be the identity. Hence, we have $P'(I) = I_1 = P'(I_1)$, yet $u(I) \neq u(I_1)$, so $P$ does not correctly express $u$.

For $\mathcal{UL}(\mathsf{delete})$, let $u$ be the update expressed by the following program:

if $T \neq \emptyset \wedge T \subseteq R \wedge T \subseteq S$ then
  $\mathsf{delete}_R(T)$;
  $\mathsf{delete}_S(T)$
else *(do nothing)* endif

This update cannot be expressed by any straight-line program $P$ using deletions only. In proof, consider the input instance $I = \{(R, \{a, b\}), (S, \{a, b\}), (T, \{b\})\}$ with $u(I) = \{(R, \{a\}), (S, \{a\}), (T, \{b\})\}$. Since each deletion can change one relation only, there is a prefix $P'$ of $P$ such that $P'(I)$ equals either $I_1 = \{(R, \{a\}), (S, \{a, b\}), (T, \{b\})\}$ or $I_1 = \{(R, \{a, b\}), (S, \{b\}), (T, \{b\})\}$. Say it is $I_1$; the argument for $I_2$ is analogous. Since $u(I_1) = I_1$, and we have only deletions, every step of $P$ on $I_1$ should be the identity. Hence, we have $P'(I) = I_1 = P'(I_1)$, yet $u(I) \neq u(I_1)$, so $P$ does not correctly express $u$.

For $\mathcal{UL}(\mathsf{modify})$, let $u$ be the update expressed by the following program:

if *($T$ is a singleton disjoint from $R$ and $S$)* then
  $\mathsf{modify}_R(\{(x, y) \mid R(x) \wedge T(y)\})$;
  $\mathsf{modify}_S(\{(x, y) \mid S(x) \wedge T(y)\})$
else *(do nothing)* endif

This update cannot be expressed by any straight-line program $P$ using only $\emptyset$-generic modifications. In proof, consider the input instance $I = \{(R, \{a_1, a_2\}), (S, \{a_1, a_2\}), (T, \{b\})\}$ with $u(I) = \{(R, \{b\}), (S, \{b\}), (T, \{b\})\}$. By symmetry, the only three possible results of a modification applied to $I$ are $I$ itself, $I_1 = \{(R, \{b\}), (S, \{a_1, a_2\}), (T, \{b\})\}$, and $I_2 = \{(R, \{a_1, a_2\}), (S, \{b\}), (T, \{b\})\}$. Since $u(I) \neq I$, there exists a prefix $P'$ of $P$ such that $P'(I)$ equals either $I_1$ or $I_2$. Say it is $I_1$; the argument for $I_2$ is analogous. The only two possible results of a modification applied to $I_1$ are $I_1$ itself and $I_3 = \{(R, \{b\}), (S, \{b\}), (T, \{b\})\}$. On $I_3$, no modifications except for the identity are possible. Since $u(I_1) = I_1$, every step of $P$ on $I_1$ should be the identity. Hence, we have $P'(I) = I_1 = P'(I_1)$, yet $u(I) \neq u(I_1)$, so $P$ does not correctly express $u$. □

*Proof of Theorem 5.1.* In this proof it will be understood that all updates considered are $\emptyset$-generic. Also, for notational convenience, we will identify an instance $I$ of $\mathbf{S}_{\text{graph}}$ with the binary relation $I(R)$. We will use unordered pairs $\{a, b\}$ as undirected edges, which should be understood as an abbreviation for the two symmetric directed edges $(a, b)$ and $(b, a)$.

To separate insert; delete from delete; insert, we use the following update $u$:

$\mathsf{insert}_R(\{(y, x) \mid R(x, y) \wedge R(y, y)\})$;
$\mathsf{delete}_R(\{(x, x) \mid R(x, x)\})$.

To show that this update cannot be expressed in the form delete; insert, consider the instance $I = \{(1, 2), (2, 2), (3, 4)\}$. We have $u(I) = \{(1, 2), (2, 1), (3, 4)\}$. The delete step must delete $(2, 2)$ and cannot delete any other edges as this would result in the loss of domain elements. Hence the delete step yields the intermediate result $J = \{(1, 2), (3, 4)\}$. By symmetry, it is impossible to map $J$ to $u(I)$ by a generic update.

To separate delete; insert from insert; delete, we use the following update $u$:

$\mathsf{delete}_R(\{(x, y) \mid R(x, y) \wedge (R(x, x) \vee R(y, y))\})$;
$\mathsf{insert}_R(\{(x, x) \mid \exists y (R(x, y) \vee R(y, x))\})$.

To show that this update cannot be expressed in the form insert; delete, consider the instance $I = \{1, 2, 3\}^2 - \{(2, 2), (3, 3)\}$. We have $u(I) = \{(2, 2), (2, 3), (3, 2), (3, 3)\}$. The insert step must insert $(2, 2)$ and $(3, 3)$, yielding the intermediate result $J = \{1, 2, 3\}^2$. By symmetry, it is impossible to map $J$ to $u(I)$ by a generic update.

To separate modify; delete from delete; modify, we use the following update $u$:

1. For each induced subgraph isomorphic to $G = \{(0,1),(0,2),(0,3),\{0,4\},\{0,5\},\{0,6\}\}$ reverse the directed edges $(0,1)$, $(0,2)$ and $(0,3)$;

2. Delete the edges from any induced subgraph isomorphic to $\{\{0,4\},\{0,5\},\{0,6\}\}$.

To show that this update cannot be expressed in the form delete; modify, consider the instance $I = G$. We have $u(I) = \{(1,0),(2,0),(3,0)\}$. We analyze what the delete step can do. Note that it has to treat $(0,1)$, $(0,2)$ and $(0,3)$ similarly by symmetry; furthermore, it has to treat $(4,0)$, $(5,0)$ and $(6,0)$ similarly, and also $(0,4)$, $(0,5)$ and $(0,6)$ similarly.

- It cannot delete the edges $(0,1)$, $(0,2)$ and $(0,3)$ as this would result in the loss of active domain elements 1, 2 and 3.
- It cannot delete both directions of the undirected edges $\{0,4\}$, $\{0,5\}$ and $\{0,6\}$, as this would result in the intermediate result $G_1 = \{(0,1),(0,2),(0,3)\}$. But on input $G_1$ itself, the intermediate result must also be $G_1$ (otherwise, the whole active domain would be lost). However, $G_1 = u(G_1) \neq u(G)$, which is a contradiction (same intermediate result but different final results).
- It cannot delete the edges $(0,4)$, $(0,5)$ and $(0,6)$ and keep the other directions, as this would result in the intermediate result $G_2 = \{(0,1),\dots,(6,0)\}$. Again we obtain a contradiction since on input $G_2$ itself, the intermediate result must also be $G_2$, yet $u(G_2) \neq u(G)$.
- Finally it cannot delete the edges $(4,0)$, $(5,0)$ and $(6,0)$ and keep the other directions, as this would result in the intermediate result $G_3 = \{(0,1),\dots,(0,6)\}$. By symmetry, it is impossible to go from $G_3$ to the $u(G)$ with a generic update.

We conclude that the delete step on input $G$ yields $G$ itself as intermediate result. By Lemma 3.4, we cannot go from $G$ to $u(G)$ by a generic modification, since for each edge $e$ involving 4, 5 or 6, there exists no pair $e'$ that is fixed with respect to $(G,e)$. Hence these edge $e$ cannot be modified.

To separate delete; modify from modify; delete, we use the following update $u$:

1. In any isolated subgraph isomorphic to the graph $G$ from the previous case, delete $\{0,4\}$, $\{0,5\}$ and $\{0,6\}$;

2. In any isolated subgraph of the form $G_1 = \{(0,1),(0,2),(0,3)\}$, reverse all edges.

To show that this update cannot be expressed in the form modify, delete, consider again the instance $I = G$. We have $u(I) = \{(1,0),(2,0),(3,0)\}$. Let us denote the modification step by $m$ and investigate how $m$ can modify the edges of $G$. As in the previous case, it has to treat the three groups of edges uniformly.

The modification must produce at least the new edges $(1,0)$, $(2,0)$ and $(3,0)$, as these cannot be produced by the subsequent deletion. By Lemma 3.4, these new edges can come only from modifying the original edges $(0,1)$, $(0,2)$ $(0,3)$, since these are the only edges with respect to which the new edges are fixed.

For the edges in each of both groups $(0,4)..(0,6)$ and $(4,0)..(6,0)$, by Lemma 3.4, there are only four possibilities for the modification:

A. no change;

B. reverse;

C. modify to $(0,0)$;

D. modify to loop at 4, 5, or 6, respectively.

So we must consider all 16 combinations of cases A–D for group $(0,4)..(0,6)$ with cases A–D for group $(4,0)..(6,0)$. First we state a trivial but generally helpful lemma.

**Lemma A.1.** *For any modification $m$ and any instance $I$, if $I \subseteq m(I)$, then $I = m(I)$.*

*Proof of Lemma.* Since $I \subseteq m(I)$, the total number of tuples in $m(I)$ is at least that of $I$. But a modification can never strictly increase the total number of tuples. Hence, $m(I) = I$. □

Now to the 16 cases.

AA The intermediate result $I'$ of the modification is like $G$ except that $(0,1)..(0,3)$ have been reversed. Note that $u(I') = I'$. Hence, since the second step is a deletion, $I'$ must be a subset of $m(I')$, whence, by the Lemma, $m(I') = I'$. Since $u(I) \neq u(G)$, however, we get a contradiction (same intermediate result but different final results).

In all the following cases we will have a similar reasoning: we obtain an intermediate result $I'$ with the two properties that $u(I') = I'$ (whence $I' = m(I')$) but $u(I') \neq u(G)$, yielding a contradiction. So, we will just list the intermediate result in each case.

AB $\{(1,0)..(3,0),(0,4)..(0,6)\}$.

AC $\{(1,0)..(3,0),(0,4)..(0,6),(0,0)\}$.

AD $\{(1,0)..(3,0),(0,4)..(0,6),(4,4)..(6,6)\}$.

BA $\{(1,0)..(3,0),(4,0)..(6,0)\}$.

BB $\{(1,0)..(3,0),(0,4)..(0,6),(4,0)..(6,0)\}$.

BC $\{(1,0)..(3,0),(4,0)..(6,0),(0,0)\}$.

BD $\{(1,0)..(3,0),(4,0)..(6,0),(4,4)..(6,6)\}$.

CA Same as BC.

CB Same as AC.

CC $\{(1,0)..(3,0),(0,0)\}$.

CD $\{(1,0)..(3,0),(0,0),(4,4)..(6,6)\}$.

DA Same as BD.

DB Same as AD.

DC Same as CD.

DD $\{(1,0)..(3,0),(4,4)..(5,5)\}$.

To separate $\mathsf{insert};\mathsf{modify}$ from $\mathsf{modify};\mathsf{insert}$, we use the following update $u$:

$$\mathsf{insert}_R(\{(x,x) \mid \exists u,v,u'(R(u,v) \wedge R(u',v) \wedge u \neq u' \wedge (x = u \vee x = v))\});$$
$$\mathsf{modify}_R(\{(x,y,y,x) \mid R(x,x) \wedge R(x,y) \wedge R(y,y)\}).$$

To show that this update cannot be expressed in the form $\mathsf{modify};\mathsf{insert}$, consider the instance $I = \{(1,0),(2,0)\}$. We have $u(I) = \{(0,1),(0,2),(0,0),(1,1),(2,2)\}$. In principle, there are six possibilities for the modification step $m$ to modify both edges $(1,0)$ and $(2,0)$ uniformly:

A. no change;

B. reverse;

C. modify to the other edge;

D. modify to loop at 0;

E. modify to loop at tail;

F. modify to loop at tail of the other edge.

Nevertheless, C has the same global effect as A, so we ignore it; D loses domain elements 1 and 2; E and F lose domain element 0. So it remains to consider cases A and B.

In case A, the intermediate result is $G$ itself. By the subsequent insertion we cannot go from $G$ to $u(G)$ because the edge $(1,0)$ of $G$ is not in $u(G)$.

In case B, the intermediate result is $I' = \{(0,1),(0,2)\}$. Note that $u(I') = I' \neq' u(G)$. We now reason that $m(I')$ must be $I'$ itself, from which we get the desired contradiction. So, suppose $m(I') \neq I'$. Then reasoning as before, the remaining possibilities for $m(I')$ are $\{(0,0)\}$, $\{(1,1),(2,2)\}$, or $\{(1,0),(2,0)\}$. The first two possibilities do not have all required domain elements, and the last possibilities cannot reach $I'$ by an insertion.

To separate $\mathsf{modify};\mathsf{insert}$ from $\mathsf{insert};\mathsf{modify}$, we use the following update $u$:

$$\mathsf{modify}_R(\{(x,y,x,x) \mid R(x,x) \wedge R(x,y) \wedge R(y,y)\});$$
$$\mathsf{insert}_R(\{(x,x) \mid \exists y(R(x,y) \vee R(y,x))\}).$$

To show that this update cannot be expressed in the form $\mathsf{insert};\mathsf{modify}$, consider the instance $I = \{1,2,3,4\}^2 - \{(3,3),(4,4)\}$. We have $u(I) = \{1,2,3,4\}^2 - \{(1,2),(2,1)\}$. On input $I$, by symmetry, there are only two possibilities for the insertion step: insert nothing, or insert $\{(3,3),(4,4)\}$. The latter possibility can be ruled out, however, because it results in the complete directed graph from which $u(I)$ cannot be produced by a generic update.

So, we are left with the modification step which acts directly on $I$, and we must show that no generic modification can map $I$ to $u(I)$. In $I$ we can distinguish several groups of edges whose members must be treated uniformly by symmetry. Two such groups are the "top loops:" the loops at 1 and 2, and the "top edges": the two edges between 1 and 2. It is very important to note that $I$ has 14 edges and $u(G)$ has 14 edges as well. Hence, the modification must be injective on the edges.

Let us see what the modification can do with a top loop. The possibilities are the following:

L1. no change;

L2. modify to the other top loop;

L3. modify to the leaving top edge;

L4. modify to the entering top edge.

By Lemma 3.4, there are no other possibilities, primarily due to the automorphism that fixes 1 and 2 but swaps 3 and 4. We can rule out possibilities L3 and L4 because they would lead to the continued presence of the top edges in the final result $u(I)$, which is incorrect. Possibilities L1 and L2 have the same global effect.

The possibilities for the modification of a top edge are the following:

T1. no change;

T2. modify to the other top edge;

T3. modify to the loop at the tail;

T4. modify to the loop at the head.

We can rule out T1 and T2 because they keep the top edges in the final result. Possibilities T3 and T4 have the same global effect.

Since the modification maps both the set of top loops, and the set of top edges, to the set of top loops, the modification is not injective on the edges and we obtain that $u(I)$ cannot be reached, as desired. $\square$