Dynamic Complexity under Definable Changes

THOMAS SCHWENTICK, TU Dortmund University, Germany
NILS VORTMEIER, TU Dortmund University, Germany
THOMAS ZEUME, TU Dortmund University, Germany

In the setting of dynamic complexity, the goal of a dynamic program is to maintain the result of a fixed query for an input database which is subject to changes, possibly using additional auxiliary relations. In other words, a dynamic program updates a materialized view whenever a base relation is changed. The update of query result and auxiliary relations is specified using first-order logic or, equivalently, relational algebra.

The original framework by Patnaik and Immerman only considers changes to the database that insert or delete single tuples. This article extends the setting to *definable changes*, also specified by first-order queries on the database, and generalizes previous maintenance results to these more expressive change operations. More specifically, it is shown that the undirected reachability query is first-order maintainable under single-tuple changes and first-order defined insertions, likewise the directed reachability query for directed acyclic graphs is first-order maintainable under insertions defined by quantifier-free first-order queries.

These results rely on *bounded bridge properties* which basically say that, after an insertion of a defined set of edges, for each connected pair of nodes there is some path with a bounded number of new edges. While this bound can be huge in general, it is shown to be small for insertion queries defined by unions of conjunctive queries. To illustrate that the results for this restricted setting could be practically relevant, they are complemented by an experimental study that compares the performance of dynamic programs with complex changes, dynamic programs with single changes, and with recomputation from scratch.

The positive results are complemented by several inexpressibility results. For example, it is shown that – unlike for single-tuple insertions – dynamic programs that maintain the reachability query under definable, quantifier-free changes strictly need update formulas with quantifiers.

Finally, further positive results unrelated to reachability are presented: it is shown that for changes definable by parameter-free first-order formulas, all LOGSPACE-definable (and even AC^1-definable) queries can be maintained by first-order dynamic programs.

CCS Concepts: • **Theory of computation → Finite Model Theory**; *Complexity theory and logic*; • **Information systems →** *Data management systems*;

Additional Key Words and Phrases: Dynamic descriptive complexity, SQL updates, dynamic programs

ACM Reference Format:
Thomas Schwentick, Nils Vortmeier, and Thomas Zeume. 2018. Dynamic Complexity under Definable Changes. *ACM Trans. Datab. Syst.* 1, 1 (July 2018), 37 pages. https://doi.org/0000001.0000001

The authors acknowledge the financial support by DFG grant SCHW 678/6-2. The third author acknowledges the financial support by the European Research Council (ERC), grant agreement No 683080, and thanks Mikołaj Bojanczyk as well as the Simons Institute for the Theory of Computing for hosting him and providing excellent research conditions.
Authors' addresses: Thomas Schwentick, TU Dortmund University, Faculty for Computer Science, Otto-Hahn-Straße 12, Dortmund, 44227, Germany, thomas.schwentick@tu-dortmund.de; Nils Vortmeier, TU Dortmund University, Faculty for Computer Science, Otto-Hahn-Straße 12, Dortmund, 44227, Germany, nils.vortmeier@tu-dortmund.de; Thomas Zeume, TU Dortmund University, Faculty for Computer Science, Otto-Hahn-Straße 12, Dortmund, 44227, Germany, thomas.zeume@tu-dortmund.de.

0362-5915/2018/7-ART $15.00
https://doi.org/0000001.0000001

1 INTRODUCTION

Modern data management systems need to handle large data sets that are subjected to frequent changes. A connection in a train network might be closed on short notice; a server might fail, leading to the cancellation of all connections to adjacent servers; or a person might add all guests of a party she visited to her friend list in a social network. How to answer queries on such dynamic data sets is a fundamental and well-studied question. Often query re-evaluation from scratch is not possible due to the size of the involved data. A standard approach to this challenge is to store materialized auxiliary data with the intention that (1) it can be used to answer a set of queries of interest, and (2) it can be updated in an efficient way. Since the stored auxiliary data can be seen as materialized views, this approach is sometimes called *incremental view maintenance* [24].

From a high level perspective, one can distinguish an algorithmic and a declarative approach towards incremental view maintenance. The former asks which algorithmic resources are necessary to update views after data has changed [30, *View Maintenance* and *Maintenance of Recursive Views*]. This approach largely ignores the kinds of operations that are directly supported by the DBMS at hand and considers "arbitrary" algorithms. It is closely related to the field of *Dynamic Algorithms* [8].

In the second approach, on the other hand, the query and data management facilities of the DBMS are taken into account and one studies how and when updates of the auxiliary data can be specified in a declarative fashion [30, *Incremental Computation of Queries*]. The idea is to avoid additional (user-defined) programs or functions and to delegate the search for a good execution strategy to the DBMS. In the case of relational databases, it is natural to choose the triumvirate of (core) SQL, the relational algebra, and the relational calculus for the specification of updates, their translation into query plans and the study of their expressive power.

Besides avoiding the recomputation of queries from scratch, the incremental maintenance of views (or query results) has an additional advantage: it allows to answer queries that are beyond the expressive power of the relational algebra, for instance certain recursive queries.[1]

The theoretical study of the declarative approach was started by Dong and Topor [17]. Dong, Su and Topor then proposed *FOIES*, a framework for incremental query evaluation based on first-order logic [12, 16]. A slightly different framework was suggested by Patnaik and Immerman [33, 34]. In both frameworks, for a given query q, one specifies a set of (logical) auxiliary relations, the *auxiliary database*, and a set of first-order update queries for each change. When the underlying database is changed, the first-order update formulas are evaluated on the auxiliary database and the underlying database to produce the updated auxiliary database.

The main difference between the two frameworks is that in the *FOIES* framework, the domain of the database can change, whereas in the framework of Patnaik and Immerman it can not. Although it is reasonable to assume that the domain can change, the other framework arguably yields a simpler setting. Furthermore, both positive and negative (inexpressibility) results usually easily translate from one setting to the other. We therefore adopt the framework of Patnaik and Immerman, for which the class of queries maintainable by first-order formulas is called DynFO. We refer to this framework as *Dynamic Descriptive Complexity*. However, we will also discuss how our positive results translate to the FOIES setting. We recall that first-order logic could be readily replaced by the relational algebra without changing the class DynFO. In some examples, we will actually use SQL to specify updates.

Example 1.1. Suppose G is a directed graph with edge relation E and the auxiliary relation T contains its transitive closure. If an edge (u, v) is inserted into E, the updated transitive closure

[1] Since the SQL:1999 standard, SQL supports recursive queries. How recursion can actually be used in queries varies over different database systems [35, 39].

contains all tuples (x, y) that satisfy the formula $T(x, y) \vee \Big(T(x, u) \wedge T(v, y)\Big)$. Thus the transitive closure of a graph can be maintained in DynFO under single-edge insertions.

In an SQL fashion[2] this update rule for the transitive closure relation can be expressed as follows:

```
ON INSERT E(u,v) UPDATE T AS
SELECT *
FROM   T
  UNION
SELECT T1.x, T2.y
FROM   T as T1, T as T2
WHERE  T1.y = u AND T2.x = v
```

The focus of Dynamic Descriptive Complexity so far has been on the ability of DynFO to maintain queries under single-tuple insertions and deletions. In recent years, this area has seen considerable progress thanks to some powerful new techniques. In particular, it has been shown that the reachability query can be maintained in DynFO for directed graphs under insertions and deletions of edges [4]. It is well-known that first-order logic can not express the reachability query (even not on simple paths), therefore the dynamic setting increases the power of first-order logic significantly.

While the study of single-tuple changes is justified by the necessity to gain a basic understanding of the dynamic setting, it is evident that more complex change operations are more interesting from a practical perspective. Of course, "arbitrary" changes of the underlying database may render the stored auxiliary relations useless. Complex changes therefore need to be restricted in some way, e.g. by size [7], by the structure of the tuples to be changed [11, 16, 40], or by considering only changes specified in some declarative language. The latter option has not been studied in-depth so far.

The aim of this article is to contribute a formalization of declaratively defined changes in the context of Dynamic Descriptive Complexity, and to explore which queries can be maintained under complex change operations.

The formal extension of the single-tuple-change paradigm to declaratively defined changes studied here is inspired by SQL update queries (for a theoretical study of SQL updates we refer to [2]). Changes to the underlying database are specified by *replacement queries* which can modify several relations at a time and are defined by first-order formulas that can use tuples of elements as parameters. Again, as first-order logic corresponds to the relational algebra core of SQL, this is a natural setting in the context of relational databases to start from. Several articles in Dynamic Descriptive Complexity proposed to study first-order defined changes [18, 23, 34]. Similar but weaker frameworks were introduced in [25, 42], but maintainability under complex changes has not been studied there.

Contributions. We mainly consider the reachability query under complex insertions and single-tuple deletions and show that the known results for single-tuple changes can be considerably extended. We show that for insertions defined by unions of conjunctive queries maintaining reachability of undirected graphs and DAGs is actually feasible, and we underpin these theoretical findings with some encouraging experiments. We complement these positive results by some negative results in the form of inexpressibility statements. Finally, we exhibit another set of positive results, for parameter-free change operations, where all LOGSPACE-definable (and even AC^1-definable) queries can be maintained by first-order dynamic programs.

[2]We note that the first line is not part of the SQL standard.

The generalized setting yields a huge range of research questions, e.g., all previously studied questions in Dynamic Desriptive Complexity in combination with replacement queries of varying expressiveness. Naturally, this article can only start to investigate a few of them.

We present positive and negative results. In Section 4 we study first-order definable *insertion* queries (supplementing the single-tuple changes). It turns out that the undirected reachability query q_{UReach} can still be maintained in DynFO under first-order definable insertions (Theorem 4.2) and the directed reachability query q_{Reach} for directed acyclic graphs under quantifier-free insertions (Theorem 4.5). These results are shown with the help of a simple concept, *bounded bridge distance*. In a nutshell, a change operation has bounded bridge distance, if whenever a graph resulting from such an operation has a path from some node u to some node v, it also has such a path that uses only a bounded number of newly inserted edges. The size of the formulas in our dynamic programs depends on the actual bridge bound and, in general, might grow non-elementarily[3]. Although in practical examples these worst-case sizes might not occur, we also study update operations of small bridge bound size, in particular, unions of conjunctive queries (UCQs). Our presentation in Section 4 assumes the bounded bridge properties as given and we prove them later, in Section 6.

We report on experiments with a prototypical implementation of the resulting dynamic programs for the undirected reachability query in Section 5. We compare the performance of the dynamic program for UCQ-defined changes with two other declarative approaches: (a) using recursion capabilities of SQL, and (b) repeated application of a dynamic program for single-edge changes (see also [32]). The dynamic program for complex changes performs best in almost all test cases, often by one or two orders of magnitude. It also still performs reasonably well on a graph with nearly two million nodes obtained from the DBLP dataset. In addition we compare the dynamic program with a problem-specific algorithm written in a general purpose programming language that computes the query result from scratch. It is to be expected that such a problem-tailored algorithm can achieve better results. Surprisingly, the performance gap is not huge, and for some test cases the dynamic program for complex changes even performs slightly better.

It is not surprising that complex changes may make maintaining a query more difficult or even impossible. However, it is notoriously difficult to actually *prove* inexpressibility results in the dynamic setting. In Section 7 we confirm the above expectation by exhibiting inexpressibility results that show that certain positive maintainability results of the single-tuple setting do not survive in the complex setting. As an example, in the setting of single-tuple insertions, reachability on directed graphs can even be maintained with quantifier-free formulas (cf. Example 1.1). We show that this is no longer the case for complex insertions defined by very simple quantifier-free insertion queries.

Section 8 explores complex changes in an entirely different direction and with very different techniques. We show that *all* queries that can be evaluated in logarithmic space (and even in uniform AC^1) can be maintained in DynFO under (finite sets of) first-order definable parameter-free replacement queries (Theorem 8.1).

Related work. Several other prior results for Dynamic Complexity under more general changes have been obtained. The reachability query for directed graphs has been studied under deletions of sets of edges and nodes that form an anti-chain in [11] and under insertions of sets of tuples that are cartesian-closed in [16]. The maintenance procedure for this query under single-tuple changes from [4] can deal with small changes, i.e., of size logarithmic in the number of nodes, to the set of outgoing edges of a node (or, alternatively, the set of incoming edges). Further generalizations of this result to changes of non-constant size are studied in [7]. Edge contractions have been studied in [40]. Koch considered more general sets of changes in [27], though only for non-recursive queries.

[3] A function is *non-elementary*, if its growth can not be bounded by a fixed-height tower of exponential functions.

The only other implementations of dynamic complexity that we are aware of are described in [32] and [27].

This article is based on the results of [36] but is considerably extended by new positive results for UCQ-defined insertions, an experimental evaluation, new lower bounds (Theorems 7.1 (b) and Theorem 7.3 (b)), and more proof details.

2 PRELIMINARIES: DATABASES, QUERIES, AND LOGIC

In this section we recall basic notions from database theory and logics. In the interest of readability we keep the formalism simple. We refer to [29] for a more detailed introduction.

A *(relational) schema* τ consists of a set of relation symbols[4]. In this work, a *domain* is a finite set. A *database* $\mathcal{D}$ over schema τ with domain D has, for every relation symbol $R \in \tau$, a relation over D. A *k-ary query* q on τ-databases is a mapping that assigns a subset of D^k to every τ-database over domain D and commutes with isomorphisms. For a first-order formula $\varphi(\bar{x})$, we write $\mathcal{D} \models \varphi(\bar{a})$ if the tuple $\bar{a}$ over domain D satisfies $\varphi(\bar{x})$.

The focus of this article is on the dynamic behaviour of graph queries. We represent graphs as databases over the schema with a single binary relation symbol E. All graphs in this article are *directed*, that is, they are of the form $G = (V, E)$ where $E \subseteq V \times V$. The *undirected graph* induced by G is the graph with the same set of nodes and edges $\{(u, v), (v, u) \mid (u, v) \in E\}$. A *path* from $u \in V$ to $v \in V$ is a sequence $u = u_0, u_1, \ldots, u_n = v$ of pairwise distinct nodes from V such that $(u_i, u_{i+1}) \in E$ for all $0 \leq i < n$. An *undirected path* in G is a path in its induced undirected graph. A *connected component* of G is a maximal connected subgraph H, i.e., for all nodes u and v of H, there is a path from u to v. A *weakly connected component* of G is a maximal subgraph whose induced undirected graph is connected. A *directed tree* is a graph with a distinguished node r (the *root* node) such that all nodes can be reached from r by exactly one (directed) path. A disjoint union of directed trees is called *directed forest*.

The *reachability query* q_{Reach} selects, given a graph G, all pairs (u, v) such that there is a path from u to v in G. Similarly, the *undirected reachability query* q_{UReach} selects (u, v) if there is an undirected path from u to v.

3 DYNAMIC PROGRAMS WITH COMPLEX CHANGES

In this section we lift the definitions from [38] to more general change operations. We first define (general) change operations, then we adapt the definition of dynamic programs of [38] to those more complex changes.

3.1 Change Operations

As mentioned before, in most investigations of DynFO, only single-tuple insertions and deletions were considered as change operations. In their most general form, the complex change operations that we consider in this article can modify a given database by replacing some of its relations with the results of first-order-defined queries on the database. Before we fix our notation of complex changes, we illustrate them by a few examples.

Example 3.1. (a) Actually, a single-tuple insertion is just a special case of a complex change operation. It only concerns one relation of the database, but it uses *parameters*. As an example, consider the change operation that allows to insert edges into the edge relation E of a graph. It uses two variables, say u and v, as parameters, representing the two affected nodes. The corresponding replacement rule $\mu_E(u, v; x, y)$ could be specified by the simple formula $E(x, y) \vee$

[4]For simplicity of the formalism, we do not denote arity functions for relation symbols. We also do not need constant symbols in this article.

$(x = u \wedge y = v)$ stating that after the operation all previous edges are still there and a (possibly but not necessarily new) edge (u, v) has been added. This operation can then be instantiated by two concrete elements a and b of the domain, for u and v.
In general, a change operation consists of a tuple of *parameters* and a set of replacement queries that might use these parameters.

(b) Suppose that a user wants to connect a node $u \in V$ to every other node of a directed graph (V, E). This can be achieved by the change operation with replacement rule $\mu_E(u; x, y) = E(x, y) \vee (x = u)$. Again, the parameter u can be instantiated with each element of the domain. This change can also be expressed by the following SQL statement[5] (ignoring duplicated insertion of edges):

```
INSERT INTO E
  SELECT u as x, V.v as y
  FROM   V
```

(c) The following example allows to change two relations by one operation. Here, we consider directed graphs with two additional unary relations C_1, C_2, which we interpret as a colouring of the nodes. Let $G = (V, E, C_1, C_2)$ be such a coloured graph and suppose a user wants to swap the colours C_1, C_2 for all nodes that have an edge to a node $u \in V$. This can be achieved by the rules **replace** C_1 **by** $\mu_{C_1}(p; x)$ and **replace** C_2 **by** $\mu_{C_2}(p; x)$ with $\mu_{C_1}(p; x) = \Big(\neg E(x, p) \wedge C_1(x)\Big) \vee \Big(E(x, p) \wedge C_2(x)\Big)$ and $\mu_{C_2}(p; x) = \Big(\neg E(x, p) \wedge C_2(x)\Big) \vee \Big(E(x, p) \wedge C_1(x)\Big)$. We emphasize that these two rules specify *one* change operation.

(d) Finally, we consider an example of a parameter-free insertion. It states that, in a graph, all nodes u and v that are connected by a path of length 2 should be connected directly. The replacement rule $\mu_E(; x, y)$ can be phrased as $\exists z(E(x, z) \wedge E(z, y)$.

More formally, a *replacement rule* ρ_R for relation R is of the form **replace** R **by** $\mu_R(\bar{p}; \bar{x})$. Here, R is a relation symbol and $\mu_R(\bar{p}; \bar{x})$ is a first-order formula, where the tuple $\bar{x}$ has the same arity as R and $\bar{p}$ is another tuple of variables, called the *parameter tuple*. A *replacement query* $\rho(\bar{p})$ is a *set* of replacement rules for distinct relations with the same parameter tuple $\bar{p}$. In the case of replacement queries ρ that consist of a single replacement rule, we usually do not distinguish between ρ and its single replacement formula μ_R.

For a database $\mathcal{D}$, a change operation $\delta = (\rho, \bar{a})$ consists of a replacement query and a tuple of elements of (the domain of) $\mathcal{D}$ with the same arity as the parameter tuple of ρ. We often use the more concise notation $\rho(\bar{a})$ and refer to change operations simply as *changes*. The result $\delta(\mathcal{D})$ of an application of a change operation $\delta = (\rho, \bar{a})$ to a database $\mathcal{D}$ is defined in a straightforward way: each relation R in $\mathcal{D}$, for which there is a replacement rule ρ_R in ρ, is replaced by the relation resulting from evaluating μ_R, that is, by $\{\bar{b} \mid \mathcal{D} \models \mu_R(\bar{a}; \bar{b})\}$.

Some of our investigations will focus on (syntactically) restricted replacement queries that either only remove or only insert tuples to relations. For an *insertion rule* ρ_R, the replacement formula $\mu_R(\bar{p}; \bar{x})$ has the form $R(\bar{x}) \vee \varphi_R$. Similarly, *deletion rules* have replacement formulas $\mu_R(\bar{p}; \bar{x})$ of the form $R(\bar{x}) \wedge \neg\varphi_R$. In [2], the change operations *replace*, *insert*, *delete* and *modify* have been studied, in particular with respect to their expressive power. These operations are captured by our change operations[6].

Other syntactic restrictions to be studied extensively in this article are *parameter-free* replacement queries (that allow no parameters in change formulas, or equivalently, no constants in relational algebra expressions) and *quantifier-free* replacement queries (that allow only quantifier-free change

[5] We represent unary relations as tables with one attribute v and binary relations as tables with attributes x, y, all of type `int`.

[6] In [2] the domain of the database can be infinite.

formulas). A special case of quantifier-free changes are the *single-tuple changes*. We refer by **insert** $\bar{p}$ **into** R to the insertion query **replace** R **by** $\mu_R(\bar{p};\bar{x})$, where $\mu_R(\bar{p};\bar{x}) = R(\bar{x}) \vee (\bar{p} = \bar{x})$ and by **delete** $\bar{p}$ **from** R to the deletion query **replace** R **by** $\mu_R(\bar{p};\bar{x})$, where $\mu_R(\bar{p};\bar{x}) = R(\bar{x}) \wedge \neg(\bar{p} = \bar{x})$. Those two replacement queries will also be denoted as $\textsc{ins}_R$ and $\textsc{del}_R$. As mentioned before, single-tuple changes are the best studied change operations in previous work on dynamic complexity. To emphasize the difference we sometimes refer to arbitrary (not single-tuple) change operations as *complex changes*. For any schema τ we denote by Δ_τ the set of single-tuple replacement queries for the relations (with symbols) in τ. In the case of graphs, we simply write Δ_E.

3.2 Dynamic Programs

We now introduce dynamic programs, mainly following the exposition in [38]. Inputs in dynamic complexity are represented as databases over a fixed domain as defined in Section 2. Initially, such a database contains no tuples. The database is then modified by a sequence of change operations.

The goal of a dynamic program is to answer a given query for the database that results from any change sequence. To this end, the program can use an auxiliary database over the same domain. Depending on the exact setting, the auxiliary database might be initially empty or not.

A dynamic program $\mathcal{P}$ operates on an *input database* $\mathcal{I}$ over a schema τ_{in} and updates an *auxiliary database* $\mathcal{A}$ over a schema[7] τ_{aux}, both containing only tuples over a domain D, which is fixed during a computation. We call $(D, \mathcal{I}, \mathcal{A})$ a *state* and basically consider it as one database over schema $\tau_{\text{in}} \cup \tau_{\text{aux}}$. The relations of $\mathcal{I}$ and $\mathcal{A}$ are called *input and auxiliary relations*, respectively.

A *dynamic program* has a set of update rules that specify how auxiliary relations are updated after a change. An *update rule* for updating an auxiliary relation T after a replacement query $\rho(\bar{p})$ is of the form **on change** $\rho(\bar{p})$ **update** $T(\bar{x})$ **as** $\varphi_T(\bar{p},\bar{x})$ where the *update formula* φ_T is over $\tau_{\text{in}} \cup \tau_{\text{aux}}$.

The semantics of a dynamic program is as follows. When a change operation $\delta = \rho(\bar{a})$ is applied to a state $\mathcal{S} = (D, \mathcal{I}, \mathcal{A})$, then the new state $\mathcal{S}' = (D, \mathcal{I}', \mathcal{A}')$ is obtained by setting $\mathcal{I}' = \delta(\mathcal{I})$ and by defining each auxiliary relation T via $T \stackrel{\text{def}}{=} \{\bar{b} \mid (\mathcal{I}, \mathcal{A}) \models \varphi_T(\bar{a},\bar{b})\}$. For a change operation δ we denote $\mathcal{S}'$ by $\mathcal{P}_\delta(\mathcal{S})$. For a sequence $\alpha = (\delta_1, \ldots, \delta_k)$ we write $\mathcal{P}_\alpha(\mathcal{S})$ for the state obtained after successively applying $\delta_1, \ldots, \delta_k$ to $\mathcal{S}$.

A *dynamic query* is a tuple (q, Δ) where q is a query over schema τ_{in} and Δ is a set of replacement queries. The dynamic program $\mathcal{P}$ *maintains* a dynamic query (q, Δ) with k-ary q if it has a k-ary auxiliary relation Q that, after each change sequence over Δ, contains the result of q on the current input database. More precisely, for each non-empty domain D, each non-empty[8] sequence α of changes, relation Q in $\mathcal{P}_\alpha(\mathcal{S}_\emptyset)$ and $q(\alpha(\mathcal{I}_\emptyset))$ coincide. Here, $\mathcal{S}_\emptyset = (D, \mathcal{I}_\emptyset, \mathcal{A}_\emptyset)$, and $\mathcal{I}_\emptyset$ and $\mathcal{A}_\emptyset$ denote the empty input and auxiliary database over D, respectively.

The class of dynamic queries (q, Δ) that can be maintained by a dynamic program with update formulas from first-order logic is called DynFO. We also say that the query q can be maintained in DynFO under change operations Δ. The class of dynamic queries maintainable by quantifier-free update formulas is called DynProp.

Example 3.2. The update procedure for the transitive closure relation of a directed graph after an edge insertion presented in Example 1.1 can now be formalized as follows. The dynamic query $(q_{\text{Reach}}, \{\textsc{ins}_E\})$ is maintained by the dynamic program that uses one auxiliary relation T, which always contains the transitive closure of the edge relation E. Its only update rule is given by

[7] To simplify the exposition, we will usually not mention schemas explicitly and always assume that the databases we talk about are compatible with respect to the schemas at hand.

[8] This technical restriction ensures that we can handle, e.g., Boolean queries with a yes-result on empty databases without initialization of the auxiliary relations. Alternatively, one could use an extra formula to compute the query result from the auxiliary (and input) database.

the formula $\varphi_T(p_1, p_2; x, y) = T(x, y) \vee \Big(T(x, p_1) \wedge T(p_2, y)\Big)$. This proves that $(q_{\text{Reach}}, \{\text{INS}_E\})$ is in DynFO. □

3.3 Complex Change Operations and Initialization of Dynamic Programs

In the presence of complex replacement queries, the initialization of the auxiliary relations requires some attention. In the original setting of Patnaik and Immerman, the input database is empty at the beginning, and the auxiliary relations are initialized by first-order formulas evaluated on this (empty) initial input database. Since tuples can be inserted only one-by-one, the auxiliary relations can be adapted slowly and it can be ensured that, e.g., always a linear order [34] or arithmetic [18] on the active domain is available.

For complex changes, the situation is more challenging for a dynamic program: as an example, for graphs, the first change could insert all possible edges and let the graph be a complete graph of size n, if n is the size of the underlying domain. To enable the dynamic program to answer whether the graph has some property like "all nodes have even degree" after this change, it needs some suitable (often: non-empty) initial values of the auxiliary relations. Since in this paper, we are mainly interested in the *maintenance* of queries and not so much in the specific complexity of the *initialization*, we do not define variants of DynFO with different power of initialization, but rather follow a pragmatic approach: whenever initialization is required, we say that the query can be maintained *with suitable initialization* and specify in the context what is actually needed. In all cases, it is easy to see that the initialization of the auxiliary relations can be computed in polynomial time.

An alternative approach would be to restrict the semantics of replacement queries to elements of the active domain of the current database and to allow the activation of elements only via tuple insertions.

3.4 Changing the Domain

The described general framework follows [34] and thus does not allow inserting new elements into or removing existing elements from the domain. Of course, for dynamical databases in practice this is a severe restriction, and therefore an alternative formalization that allows to adapt the underlying domains has been proposed. In the first-order incremental evaluation system framework (short: FOIES) introduced by Dong, Su, and Topor [16], changes to the database may involve elements that have not been in the domain so far which are then added to the domain automatically. Apart from this difference, the first-order update mechanism of FOIES and the one described above are the same.

On a superficial level, due to the necessity to deal with changing domains, it may appear that it is more difficult to show that queries can be maintained by a FOIES. Yet, almost all results for single-tuple changes translate effortlessly between the two frameworks. A theoretical justification for this observation is provided in [5, Theorem 17]. Inexpressibility results for DynFO translate directly to FOIES.

Inserting several new elements in the domain at once poses the same problems as discussed in the previous subsection. A convenient way to extend the dynamic descriptive complexity framework to support the addition and removal of elements that circumvents such problems is to introduce two more change operations, add(x) and remove(x).

All results from Section 4 and Section 7 carry over easily for this extension. The former holds since we use the same auxiliary relations as in [15] and [13], respectively, where single-tuple maintainability was shown for FOIES. The latter holds since inexpressibility results always carry over from FOIES to DynFO. Since add(x) and remove(x) have parameters, they do not quite fit for

parameter-free changes. However, with suitable adaptations, discussed at the end of Section 8, the obtained results can be extended to dynamic domains, as well.

4 REACHABILITY AND DEFINABLE INSERTIONS

We first study the impact of first-order definable complex change operations on the (binary) reachability query. In the classical DynFO setting with single-tuple change operations it was shown early on that reachability can be maintained in DynFO for two important special cases: undirected reachability and directed reachability on acyclic graphs (dags) [13, 15, 34]. We show here that these results can be extended to complex *insertions*.

These results are shown with the help of a simple concept, *bounded bridge distance*. In a nutshell, a change operation has bounded bridge distance, if whenever a graph resulting from such an operation has a path from some node u to some node v, it also has such a path that uses only a bounded number of newly inserted edges.

The remainder of this section consists of two parts. In the first part we consider the undirected reachability query and present dynamic programs that maintain this query under first-order insertions. Unfortunately those programs are not very practical as their size is non-elementary in the size of the first-order insertions. Therefore we present more efficient programs for undirected reachability under UCQ- and $\text{UCQ}^{\neg}$-definable insertions afterwards.

In the second part we present a dynamic program for the reachability query under quantifier-free insertions on directed acyclic graphs. All these programs basically use the same auxiliary relations as in the case of single-tuple changes. Both subsections start with a more formal definition of the concept of bounded bridge distance. For each different kind of change operation we state their respective bridge distance bounds which are used by the dynamic programs.

In Section 5, we report on experiments with a prototypical implementation of the resulting dynamic programs for two CQ- and UCQ-definable change operations, respectively. The proofs of the various bounded bridge distance results are delegated to Section 6. In Section 7, we show complementing inexpressibility results for two prominent fragments of DynFO under complex changes.

4.1 Undirected Reachability

In this subsection, we show that the undirected reachability query can be maintained in DynFO under single-edge insertions and deletions and any finite set of first-order definable insertions. The respective dynamic programs rely on bounds on the undirected bridge distance, to be defined next, and the following Proposition 4.1 that will be shown in Section 6.

Let G' be a graph that is obtained from a graph G by an insertion of edges. For two nodes u, v of G', the *undirected bridge distance* $\text{ubd}_{G,G'}(u, v)$ is the minimal number d, such that there is a path from u to v in G' that uses at most d edges that are not in G. We will refer to the new edges in such paths as *bridges*. Since the two graphs will always be clear from the context, we usually simply write $\text{ubd}(u, v)$ instead of $\text{ubd}_{G,G'}(u, v)$. We say that an insertion query ρ has the *undirected bounded bridge property* if there is a constant c such that, for every graph G' resulting from a graph G by applying ρ, and all nodes u, v of G, $\text{ubd}_{G,G'}(u, v) \leq c$. The smallest such c is called the *undirected bridge bound* of ρ.

Proposition 4.1. *Every first-order definable insertion query has the undirected bounded bridge property.*

Proposition 4.1 underlines the benefits of a theoretically well-founded framework. Its proof is not very difficult with the help of some basic model-theoretic concepts such as rank-k-types. Since we

want to avoid this machinery in the early sections of this article, we postpone its proof to Section 6. With the help of Proposition 4.1, we can show the following result.

Theorem 4.2. *Let Δ be a finite set of first-order insertion queries. Then $(q_{UReach}, \Delta \cup \Delta_E)$ can be maintained in DynFO.*

For the proof, we use the approach for maintaining q_{UReach} under single-edge insertions and deletions from [15, Theorem 4.3] and maintain a directed spanning forest and its transitive closure relation. The undirected bounded bridge property allows the update of the spanning forest and its transitive closure in a first-order definable way.

Proof. The dynamic program presented in [15, Theorem 4.3] maintains the symmetric transitive closure of graphs under single-edge changes with the help of auxiliary relations H and TC_H. The binary relation H is a directed forest, whose undirected version is a spanning forest of the undirected version of the input graph G, and TC_H is its transitive closure.[9] Observe that two nodes u and v are in the same weakly connected component if and only if $\{(w,u),(w,v)\} \subseteq TC_H$ holds, for some node w.

We show how to maintain the relation H and TC_H for a single FO insertion ρ. Since Δ is finite, the update program for Δ is just the union of the update programs for each $\rho \in \Delta$. For the moment we assume a predefined linear order $\leq$ on the domain to be present. Let G be a graph and $\delta = \rho(\bar{a})$ an insertion, H a directed spanning forest of G and TC_H its transitive closure. We show how to FO-define the auxiliary relations H' and TC'_H for the modified graph $G' = \delta(G)$.

We first describe a strategy to define H' and then argue that it can be implemented by a first-order formula. We call the smallest node of a weakly connected component C' of G' with respect to $\leq$ the *queen* u_0 of C'. For each weakly connected component C of G that is a subgraph of C', we define its *queen level* as the (unique) number $\text{ubd}(u_0, u)$, for nodes $u \in C$. Thanks to Proposition 4.1 the queen level of each component is bounded by a constant c.

A directed edge (u,v) is inserted into H' if the weakly connected components of u and v have queen levels ℓ and $\ell+1$ with respect to some node u_0 and for some $\ell < c$, and (u,v) is the lexicographically smallest pair with respect to $\leq$ for which δ inserts (u,v) or (v,u). Since $\text{ubd}(u,v) \leq c$, for all nodes u, v in any component of $\delta(G)$ and because, for each number h, there are formulas $\theta_h(x,y)$ expressing that $\text{ubd}(x,y) \leq h$, the lexicographically minimal bridges can be defined by a first-order formula. In order to obtain a directed spanning forest, the direction of some edges in the directed spanning tree of C might need to be flipped, as described in [15, Lemma 4.2].

Since the construction of H' ensures that along each directed path at most c new edges are inserted, it is straightforward to extend the update formula of [15, Lemma 4.2] for $TC_{H'}$. The update formulas for the deletion of edges are the same as in the single-edge modification case.

It remains to show how the assumption of a predefined linear order can be eliminated. For a change sequence α, we denote by A_α the set of parameters used in α. When applying α to an initially empty graph, a linear order on A_α can be easily constructed as in the case of single-tuple changes [18]. The crucial observation is that the remaining nodes in $V \setminus A_\alpha$ behave identically with respect to all other nodes. More precisely, one can show by induction on $|\alpha|$, that for all nodes $a \in V$ and $b, b' \in V - A_\alpha$ it holds $(a,b) \in E \Leftrightarrow (a,b') \in E$ (and likewise $(b,a) \in E$ if and only if $(b',a) \in E$).

The dynamic program for maintaining q_{UReach} maintains the relations H and TC_H as described above, yet restricted to the induced (and ordered) subgraph G_α of G on A_α. Whether two nodes are

[9] In [15], undirected graphs are considered but it is easy to see that q_{UReach} on directed graphs can be basically maintained in the same way. Furthermore, the relation H is used slightly differently, but the adjustments are straightforward.

in the same weakly connected component of G_α can thus be inferred from H and TC_H. Whether two nodes are in the same weakly connected component of G can be easily determined as follows.

- Two nodes $a, a' \in A_\alpha$ are in the same weakly connected component of G if and only if they are in the same weakly connected component of G_α or there is a node $c \in V \setminus A_\alpha$ such that there are edges (b, c) (or (c, b)) and (b', c) (or (c, b')) connecting c with some node b in the connected component of a in G_α and some node b' in the weakly connected component of a' in G_α.
- Two nodes $a \in A_\alpha$ and $b \in V \setminus A_\alpha$ are connected by a path if and only if there is an edge between b and some node in the weakly connected component of a in G_α.
- Finally, two nodes $a, a' \in V \setminus A_\alpha$ are connected if and only they are connected by an edge or they are both connected by an edge to some node $b \in A_\alpha$.

□

The proof of Theorem 4.2 relied on the undirected bounded bridge property of first-order-definable insertions. In a slightly different dynamic setting, the result can be generalised to insertions defined by stronger logics like monadic second-order logic, as long as they have the undirected bounded bridge property. In this alternative setting, the inserted edges are given to the program not by a formula and parameters, but rather as a set of edges that has been defined by some formula.

The size of the dynamic programs of the proof of Theorem 4.2 depends on the undirected bridge bounds of the insertion queries of Δ. Unfortunately, in the worst case the bridge bound grows non-elementarily in the quantifier-rank. We therefore turn next to insertion queries with more desirable bridge bounds (and reasonably small dynamic programs). More precisely, we consider variants of conjunctive queries. We will see that for all these variants the bridge bounds – and therefore also the size of the dynamic update programs – is bounded by the size of the insertion queries.

Conjunctive queries correspond to select-project-join queries in the relational algebra and to select-from-where queries with conjunctions of atoms in where-clauses in SQL [1], and constitute one of the most investigated query languages for relational databases. We will see that the undirected bridge bound of CQ-defined insertions is at most two and, more generally, for UCQ-defined insertions based on the union of ℓ conjunctive queries it is at most 2ℓ. For CQ$^\neg$-defined insertions, there is no constant undirected bridge bound, but it is still linear in the size of the query.

More formally, the class UCQ contains all *unions of conjunctive queries* (short: *UCQs*), that is, queries expressible by formulas of the form $\varphi(\bar{w}) = \bigvee_{i=1}^{\ell} \exists \bar{z} \psi_i$ where each ψ_i is a conjunction of atomic formulas.[10] A *conjunctive query* is a UCQ for which $\ell = 1$ holds. The class of conjunctive queries is denoted by CQ. The extensions of CQ and UCQ by allowing negated atoms are denoted by CQ$^\neg$ and UCQ$^\neg$, respectively. In the following, a *UCQ-definable insertion query* is a modification query $\rho(\bar{p})$ with a formula of the form $E(x, y) \vee \varphi(\bar{p}, x, y)$, where φ is a union of conjunctive queries. We will refer to $\varphi(\bar{p}, x, y)$ as *insertion formula*. Analogously, we define *CQ-definable insertion queries* and *CQ$^\neg$-definable insertion queries*.

Bounds on the bridge bounds of UCQ-defined insertion queries are stated in the following proposition, which is proved in Section 6.

Proposition 4.3. *The following undirected bridge bounds hold.*

(a) For each UCQ-defined insertion query ρ that is a union of ℓ conjunctive queries, the undirected bridge bound of ρ is at most 2ℓ. Moreover, this bound is tight.

[10] For notational simplicity we require here that each disjunct uses the same variables.

(b) For each UCQ¬-defined insertion query ρ that is a union of ℓ conjunctive queries with negations with at most k positive atoms each, the undirected bridge bound of ρ is at most $2(k-1)\ell$.
(c) For each $n \in \mathbb{N}$ there is a CQ¬-defined insertion query ρ with undirected bridge bound $\geq n$.

In a nutshell, the reason why Statement (a) holds is that if some CQ inserts at least three bridges of a (new) path from u to v then it also has to insert some shortcut bridge.

With the bridge bounds of Proposition 4.3 the dynamic program constructed in the proof of Theorem 4.2 becomes actually usable. Indeed, Statement (a) of Proposition 4.3 is the basis for our implementation and experiments that are described in Section 5.

4.2 Reachability on Directed Acyclic Graphs

Now we turn to the other restriction for which DynFO maintainability under complex insertions (and single-edge deletions) is preserved: directed reachability on directed, acyclic graphs. However we are only able to show this result for quantifier-free insertions. In [34, Theorem 4.2], edge insertions are only allowed if they do not add cycles. Of course, given the transitive closure of the current edge relation it can be easily checked by a first-order formula (a *guard*) whether a new edge closes a cycle. We will see that this is also possible for the complex insertions we consider.

As in the previous subsection, we begin by stating a bounded bridge property. However, we have to modify it a bit to cope with the fact that insertion queries can introduces cycles into acyclic graphs. The *directed bridge distance* bd is defined as the undirected bridge distance ubd, but with respect to directed paths. We say that an insertion query ρ has the *bounded bridge property* on directed acyclic graphs, if there is a constant c such that, for every graph G' resulting from a graph G from C by applying ρ, G' contains a directed cycle with at most c bridges or, for all nodes u, v of G, $\mathrm{bd}_{G,G'}(u,v) \leq c$.

Proposition 4.4. *Every quantifier-free insertion query has the bounded bridge property on directed acyclic graphs.*

The proof is again delegated to Section 6.

This property allows extending the technique for maintaining the transitive closure relation of acyclic graphs under single-tuple changes used in [34] and [13] to quantifier-free insertions. As in [34] and [13] no further auxiliary relations besides the transitive closure relation are needed. In Section 7 we show that the transitive closure relation does *not* suffice for maintaining q_{Reach} for acyclic graphs under insertions definable with existential quantifiers.

Theorem 4.5. *Let Δ be a finite set of quantifier-free insertion queries. Then $(q_{Reach}, \Delta \cup \Delta_E)$ can be maintained in DynFO for directed, acyclic graphs. Furthermore, for each quantifier-free insertion, there is a first-order guard which checks whether the insertion destroys the acyclicity of the graph.*

Proof sketch. In [34, Theorem 4.2] and [13, Theorem 3.3], dynamic programs are given that maintain the transitive closure of acyclic graphs under single-edge modifications, using only the transitive closure as auxiliary relation. Thanks to Proposition 4.4, these programs can be easily extended. Indeed, since the number of bridges of cycles created by the insertion, and — if the graph remains acyclic — the bridge distance between two path-connected nodes are bounded by a constant, a guard formula and an update formula for the transitive closure can be constructed in a straightforward manner. □

5 CASE STUDY: IMPLEMENTING DYNAMIC UNDIRECTED REACHABILITY UNDER COMPLEX INSERTIONS

Dynamic DynFO programs can straightforwardly be implemented in relational database systems, since first-order logic is a subset of SQL with respect to expressivity. Despite this, actual implementations and evaluations are rare.

In this section we empirically compare the evaluation of the undirected reachability query q_{UReach} under first-order defined insertions of edges using

(1) dynamic programs for complex changes, following the approach of Theorem 4.2;
(2) dynamic programs that process complex changes by treating them as a sequence of single-edge changes;
(3) evaluation from scratch using SQL's recursive queries; and
(4) evaluation from scratch using standard imperative algorithms, implemented in Python.

We do *not* include graph database systems into our comparison. In a preliminary test using `Neo4j`[11] and its standard Cypher interface, query evaluation did not terminate in reasonable time even on very small instances. For general recursively defined queries a similar behavior was observed in a recent study [3]. We note that plug-ins[12] are available that enable `Neo4j` to evaluate q_{UReach} efficiently. The corresponding approach of query evaluation is, except of the chosen programming language, covered by method (4) in our comparison.

Our implementations actually can update q_{UReach} under more general first-order definable edge-insertions on node coloured graphs. All variants can also deal with single-edge deletions, yet we focus on first-order definable insertions since the (single-tuple) deletion of edges has been evaluated already in [32].

This comparison is primarily intended as a proof of concept of the results obtained in Section 4. Our results give strong arguments for the use of the dynamic approach for query evaluation. For complex changes, our implementations of dynamic programs are considerably faster than recomputation from scratch within SQL in almost all of the considered scenarios, and in some scenarios even faster than the Python-based recomputation from scratch. This was not clear to us before the empirical evaluation, since the benefit of reusing previously computed information might have been smaller than the cost of updating all auxiliary relations. Furthermore, the dynamic program for complex changes performs in general better than the dynamic program for single-edge changes which is invoked once for every changed edge. The former program is capable of dealing with a graph with nearly two million nodes obtained from the DBLP database.

Before we describe the implementation and our experiments in more detail, we quickly mention previous implementations of dynamic programs. Dong et al. present SQL code for maintaining reachability for acyclic directed, undirected, and general directed graphs under single-edge changes[13] and analyse the number of joins needed to process a single-edge change [9]. Empirical results for shortest distances in weighted undirected graphs under deletions, reported in [32], indicate that the dynamic complexity approach can outperform recomputation from scratch significantly. The compilation of a class of non-recursive queries into incremental programs has been studied and implemented in [27, 28]. Evaluations of different strategies to maintain joins of base relations are given in [41].

Next, we describe our implementation and, afterwards, we present our experiments and their results.

[11] see https://neo4j.com/
[12] e.g. https://neo4j-contrib.github.io/neo4j-graph-algorithms/
[13] The code for reachability in general directed graphs cannot be translated to a DynFO-program, as it uses exponentially many constants not present in the input graph. It further may need exponentially many updates per change step.

5.1 Implementation

We implement change operations and the evaluation methods (1)-(3) as PL/pgSQL functions for a PostgreSQL database system. PL/pgSQL[14] is a procedural language that enables the definition of functions and allows the use of control structures in addition to SQL queries. From the latter, we only use if-then-else blocks.

In our implementation, a change function writes the set of all inserted edges to a table `delta`. The PL/pgSQL function `DYN-complex`, which implements the dynamic program that processes complex changes, accesses this table and updates the auxiliary relations accordingly. It is invoked before the changes are actually applied to the database. For processing a complex change as a sequence of edge insertions, the dynamic program for single-edge insertions from [15] is used. The PL/pgSQL function `DYN-single` implementing this program is invoked by a trigger when the changes of `delta` are actually applied to the database, once for every edge. The PL/pgSQL function `STATIC-SQL` that recomputes q_{UReach} from scratch is invoked after the changes are applied and uses Common Table Expressions, more specifically the `WITH RECURSIVE` construct[15]. It also has access to `delta`, which is used, e.g., to avoid recomputation when no edge between two distinct connected components was inserted. The algorithm `STATIC-Python` computes the connected components of an undirected input graph and writes the result to the database. All four implemented functions also work if graphs are coloured, hence we use coloured graphs for some of our experiments.

In order to store the query result for q_{UReach} in a compact way, the functions use a binary table `connected_components` that contains for each node a unique representative of its weakly connected component. In the implementation, nodes are represented by integers and the representative of a component is the minimal node with respect to the natural linear order. The transitive closure of the graph is easily definable in SQL or first-order logic using this table. The reason to use `connected_components` is its succinctness: it is of linear size with respect to the number of nodes, whereas the full transitive closure might be of quadratic size.

For [15, Theorem 4.3] and Theorem 4.2, some effort is needed to maintain a linear order on the nodes. As each database comes with a natural order — in our case, the natural order on the integers —, it is not necessary to maintain a linear order. Apart from this difference and the rather straightforward maintenance of `connected_components`, the implementations of the dynamic programs follow the proofs of the corresponding results closely.

5.2 Experiments and discussions

We conducted three experiments comparing the performance of the four approaches for evaluating q_{UReach} under complex insertions. The first two experiments explore how the approaches differ depending on the number of connected components that are joined by bridges. The third experiment examines how they scale to large graphs. The experiments use insertions with small bridge bounds, following the insights from Proposition 4.3.

All experiments were conducted on a machine with 28 CPU cores (56 threads) with 2.6 GHz base frequency, of which we use only one, and 52 GB main memory, running Ubuntu 16.04, with a local default installation of PostgreSQL 11devel. `STATIC-Python` uses Python 3.5.2 and version 1.11 of the `NetworkX` package[16]. If not stated otherwise, running times are obtained from six runs per graph instance. The first run is discarded, as well as the fastest and the slowest run. The reported time is the average of the three remaining runs (cf. [3]). Individual timings include the time to update `connected_components` and, for the dynamic programs, a spanning forest and its transitive

[14] https://www.postgresql.org/docs/current/static/plpgsql-overview.html
[15] https://www.postgresql.org/docs/current/static/queries-with.html
[16] see https://networkx.github.io/

closure. Not included is the time needed to compute `delta`, to apply the changes to the database, and, for `STATIC-Python`, to build the input graph object.

First experiment. In the first experiment, we tested the evaluation methods for a change that connects many different weakly connected components of the input graph. The considered change operation inserts edges from some specified node v to all C_1 coloured nodes in a coloured graph, as expressed by $\rho_1(v) = \mu_E(v; x, y) = E(x, y) \vee \Big((x = v) \wedge C_1(y)\Big)$. This insertion query is CQ-definable, and its bridge bound is 2.We tested this change on graphs of different sizes and for a varying number of edges. For each $n \in \{10000, 20000, 30000, 40000, 50000, 75000, 100000\}$ and $p \in \{0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3\}$ we constructed five graphs with n nodes, whose colours were chosen uniformly at random from ten colours. Each graph was the disjoint union of an appropriate number of graphs with 50 nodes each. For each of these graphs, the probability of an (undirected) edge to be present was p. For each n we randomly chose a node v_n, applied the change $\rho_1(v_n)$ to each graph of size n, and measured the time it took each evaluation method to (re-)compute the table `connected_components`. Table 1 and Figure 1 give details on the graphs and the results of the experiment, where the stated values for every pair n, p average over the five instances. Because of the large running time, `STATIC-SQL` was, for $p \neq 0$, only tested for instances with at most 30000 nodes. Values marked * were obtained with a reduced testing schedule: due to large running times, only two runs on two instances were performed for every pair n, p, and the average over the four runs is reported.

We observe that in this setting, where many weakly connected components can be joined by one change, `DYN-complex` runs around two orders of magnitude faster than `DYN-single`. This is as expected, since the latter program has to update the auxiliary information for many nodes multiple times. Apart from the case $p = 0$ of initially empty graphs, direct processing of complex changes is also faster than computation from scratch using SQL. In our tests, `DYN-complex` ran three to four orders of magnitude faster than `STATIC-SQL`, which did not even terminate in reasonable time on larger graphs. For $p = 0$, `STATIC-SQL` is faster than `DYN-complex`. This is as expected: because no edge was present before the change, no prior auxiliary information is available to the dynamic programs. Basically, they also have to recompute the query result from scratch.

In all cases, the dynamic approaches are outperformed by the Python-based evaluation from scratch. For $p \neq 0$, the speed-up is between 2 and 8.

Second experiment. In the second experiment, we compared the evaluation methods for the same graphs but for a change operation that connects only few weakly connected components. More precisely, the change operation ρ_2 with parameters $v_1, \ldots, v_7$ connects all neighbours of v_1 and v_2 with the nodes $v_3, \ldots, v_7$ through the rule $\rho_2(\bar{v}) = \mu_E(v_1, \ldots, v_7; x, y) = E(x, y) \vee \Big((E(x, v_1) \vee E(x, v_2)\Big) \wedge \Big(y = v_3 \vee y = v_4 \vee y = v_5 \vee y = v_6 \vee y = v_7)\Big)$. This change can be expressed by a union of ten conjunctive queries. Its bridge bound is easily seen to be 2, which is much better than the bound of 20 given by Proposition 4.3.

The set-up of this experiment is analogous to the first experiment. The case $p = 0$ was omitted, since ρ_2 does not change empty graphs at all. The results are provided in Table 2 and Figure 2.

In this experiment, `DYN-complex` was again considerably faster than `STATIC-SQL` and even slightly faster than `STATIC-Python`. Also it outperformed `DYN-single`. This program in turn performed very well on large graphs even in comparison to `STATIC-Python`. This seems to be because it can tell rapidly whether an inserted edge lies inside a weakly connected component, and thus can be ignored. `STATIC-SQL` performed better than in the first experiment, since the weakly connected components after the change are smaller here.

Table 1. Results of the first experiment (insertion of a star into graphs with n nodes, consisting of random subgraphs of size 50 and edge probability p). The columns $|E|$ and cc's provide the number of edges and weakly connected components before the change. $\Delta|E|$ provides the number of inserted edges by the change, and Δcc's gives the difference of the number of weakly connected components. The columns DYN-c, DYN-s, STAT-SQL, STAT-Py give the runtime in seconds of the algorithms DYN-complex, DYN-single, STATIC-SQL, STATIC-Python, respectively. The values are avaraged over five instances for each pair (p, n).

p	n	$\|E\|$	cc's	$\Delta\|E\|$	Δcc's	DYN-c	DYN-s	STAT-SQL	STAT-Py	$\frac{\text{DYN-s}}{\text{DYN-c}}$	$\frac{\text{STAT-SQL}}{\text{DYN-c}}$	$\frac{\text{STAT-Py}}{\text{DYN-c}}$
0	10000	0	10000	943	-943	1.74	27.43	1.2	0.33	15.79	0.69	0.19
0	20000	0	20000	2048	-2048	9.75	118.9	4.86	0.55	12.19	0.5	0.06
0	30000	0	30000	3118	-3118	19.51	234.48	11.3	0.78	12.02	0.58	0.04
0	40000	0	40000	3970	-3970	39.06	1426.98*	18.36	1.05	36.53*	0.47	0.03
0	50000	0	50000	5038	-5038	33.19	1404.24*	31.55	1.22	42.31*	0.95	0.04
0	75000	0	75000	7490	-7490	47.47	2473.57*	69.59	1.91	52.11*	1.47	0.04
0	100000	0	100000	10120	-10120	145.93	5588.25*	130.78	2.42	38.29*	0.9	0.02
0.05	10000	12259.6	1124	998.8	-308.2	0.91	93.42	113.06	0.34	102.78	124.39	0.37
0.05	20000	24577.6	2229.8	1991.8	-600.8	2.38	402.25	487.48	0.53	168.86	204.64	0.22
0.05	30000	36835.8	3377.2	2988	-922.6	3.79	777.28	1237.56*	0.79	205.15	326.64*	0.21
0.05	40000	48991.6	4530.2	3969.6	-1230.2	7.85	2887.69*	-	0.97	367.98*	-	0.12
0.05	50000	61259	5649.8	4943	-1523.4	7.93	2867.01*	-	1.25	361.65*	-	0.16
0.05	75000	91862	8457.6	7406.2	-2289.4	12.55	5643.84*	-	1.88	449.56*	-	0.15
0.05	100000	122644	11291	9975.2	-3063.6	20.16	12227.03*	-	2.35	606.38*	-	0.12
0.1	10000	24546.4	261.4	983.2	-203	0.75	54.03	192.53	0.31	71.8	255.85	0.41
0.1	20000	48911	519.4	1988.2	-407.4	1.8	209.54	865.77	0.53	116.63	481.87	0.3
0.1	30000	73412.4	789.8	2977	-613.2	2.59	410.03	2145.46*	0.76	158.07	827.08*	0.29
0.1	40000	97919.4	1034.6	4002.4	-817.8	4.43	1432.73*	-	0.91	323.42*	-	0.2
0.1	50000	122317	1297.6	5003.8	-1023.8	5.08	1473.1*	-	1.26	289.72*	-	0.25
0.1	75000	183908.8	1923.6	7505.8	-1538	7.84	2595.37*	-	1.84	330.89*	-	0.23
0.1	100000	245126.4	2575.8	9966.8	-2043	11.24	6004.65*	-	2.50	534.15*	-	0.22
0.15	10000	36659.4	204.2	985	-198.8	0.76	46.35	248.44	0.3	60.76	325.67	0.39
0.15	20000	73491.2	406	1982	-397	1.58	179.11	1106.47	0.54	113.02	698.19	0.34
0.15	30000	110142.8	609.8	2975.4	-598	2.52	355.1	2732.98*	0.81	140.84	1083.99*	0.32
0.15	40000	147038	815.4	3996.2	-797	4.01	1253.63*	-	1.05	312.48*	-	0.26
0.15	50000	183608.2	1017.6	5027.2	-996.6	4.74	1252.73*	-	1.26	264.24*	-	0.27
0.15	75000	275649.8	1527.8	7377.8	-1491.2	7.02	2148.04*	-	1.84	306.06*	-	0.26
0.15	100000	367722.8	2037.6	10038.2	-1992.6	10.36	4390.39*	-	2.56	423.73*	-	0.25
0.2	10000	48912	200.2	1001.6	-198	0.64	42.45	301.25	0.3	66.55	472.25	0.48
0.2	20000	97934.6	400.6	2004.8	-396.4	1.48	177.88	1350.96	0.55	120.14	912.48	0.37
0.2	30000	146978.6	600.6	2991.8	-597.6	2.17	327.65	3321.56*	0.8	151.03	1531.12*	0.37
0.2	40000	195696.4	801	3984.6	-795	3.86	1189.67*	-	1.04	307.87*	-	0.27
0.2	50000	245184.2	1000.8	4975.8	-992.6	4.29	1190.05*	-	1.29	277.59*	-	0.3
0.2	75000	367597.4	1501.2	7560.6	-1493.8	6.63	2064.55*	-	1.85	311.61*	-	0.28
0.2	100000	490189.4	2002	9961.6	-1988.8	10.07	3912.13*	-	2.56	388.51*	-	0.25
0.25	10000	61323.2	200	975.4	-198.6	0.65	41.86	369.78	0.3	64.59	570.55	0.47
0.25	20000	122483.6	400	2022	-397	1.43	175.22	1549.37	0.55	122.58	1083.88	0.38
0.25	30000	183640.2	600	2993.2	-596.8	2.17	308.71	3992.07*	0.83	142.37	1841.06*	0.38
0.25	40000	244909.2	800	3977	-794	3.44	1089.95*	-	1.04	316.77*	-	0.3
0.25	50000	306185	1000	4979	-993.4	4.07	1141.95*	-	1.31	280.37*	-	0.32
0.25	75000	459241.8	1500.4	7537.2	-1493	6.77	1954.61	-	1.88	288.89*	-	0.28
0.25	100000	612608	2000	9956.4	-1987.4	9.86	3817.87*	-	2.61	387.37*	-	0.27
0.3	10000	73516.6	200	987.6	-199	0.58	40.34	417.09	0.32	69	713.43	0.54
0.3	20000	146868.2	400	1998.2	-395.8	1.36	162.49	1780.68	0.62	119.43	1308.86	0.46
0.3	30000	220461.2	600	2999	-596.6	2.13	295.67	4408.44*	0.8	139.1	2074.04*	0.38
0.3	40000	294246.4	800	3992	-793.2	3.33	1078.65*	-	1.11	323.76*	-	0.33
0.3	50000	367958.6	1000	4939.6	-994.4	4.16	1134.23*	-	1.29	272.53*	-	0.31
0.3	75000	551294.2	1500	7481.2	-1489.6	6.19	1908.72*	-	1.92	308.38*	-	0.31
0.3	100000	735921.8	2000	10018.6	-1987	9.64	3789.45*	-	2.6	393.14*	-	0.27

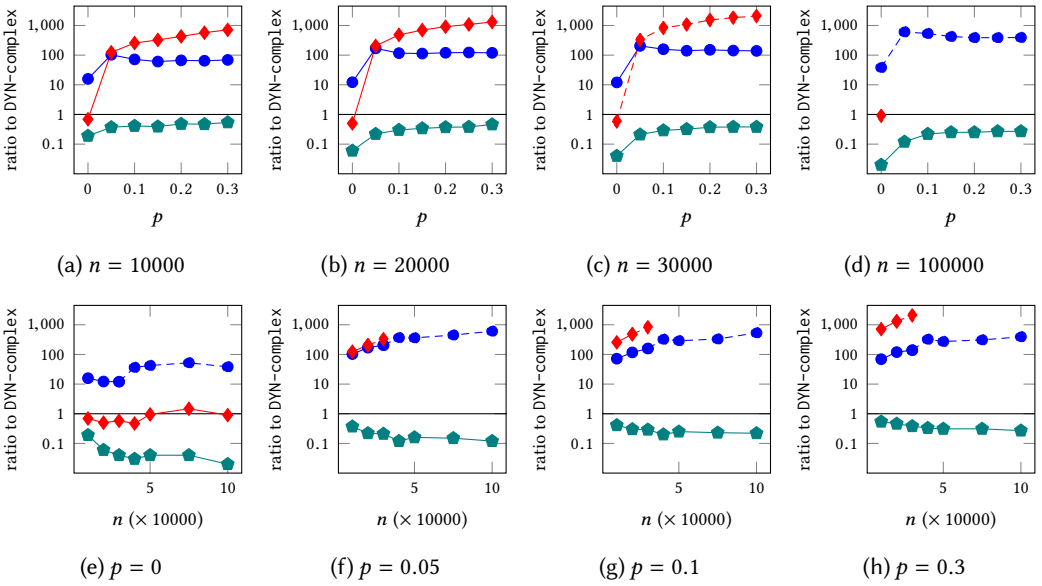

Fig. 1. Ratios of the running times in Experiment 1 of the different evaluation methods compared to DYN-complex, for different values of n and p, as listed in Table 1. Dashed lines indicate that the values are obtained with a reduced testing schedule. ●: $\frac{\texttt{DYN-single}}{\texttt{DYN-complex}}$ ♦: $\frac{\texttt{STATIC-SQL}}{\texttt{DYN-complex}}$ ⬟: $\frac{\texttt{STATIC-Python}}{\texttt{DYN-complex}}$

Third experiment. For the last experiment, we tested the performance of the dynamic programs on a large real-world graph. The graph G_{dblp} is obtained from a snapshot of the DBLP dataset from June 2017[17]. The nodes of G_{dblp} correspond to authors, and edges are based on co-authorship: an (undirected) edge (u, v) implies that the authors corresponding to u and v are co-authors of some publication.

We tested the performance for both insertions ρ_1 and ρ_2. As ρ_1 is an insertion query for coloured graphs, we coloured one out of thousand nodes with colour C_1. Details on the DBLP graph and the results of the experiment are provided in Table 3.

While DYN-complex updated the query result in a reasonable amount of time in both tests, DYN-single was only able to do so for the "easier" change ρ_2. It needed more than 100 minutes to process the change ρ_1. However, for ρ_2, DYN-single was actually the fastest method. Again, DYN-complex was slightly faster than STATIC-Python. STATIC-SQL reached its limits in this experiment: it did not finish within twelve hours.

Discussion. We conclude from the results of the experiments that using dynamic programs and in particular DYN-complex to maintain undirected reachability can reduce the running time significantly, especially compared with STATIC-SQL. Not surprisingly, a special purpose program evaluating from scratch can achieve better results. That said, we actually expected the gap to DYN-complex to be larger, and we are surprised that for some test cases, and in particular for the large real-world graph G_{dblp}, DYN-complex even performs slightly better. In any case, if no such program is at hand, the dynamic solution can be by far the best solution.

The results indicate that, as expected, the running time of DYN-single and DYN-complex is dominated by the number of connected components that are connected by the change; the running

[17] see http://dblp.dagstuhl.de/xml/release/

Table 2. Results of the second experiment (insertion of edges between up to seven connected components of graphs with n nodes, consisting of random subgraphs of size 50 and edge probability p). See Table 1 for an explanation of the columns.

p	n	$\|E\|$	cc's	$\Delta\|E\|$	Δcc's	DYN-c	DYN-s	STAT-SQL	STAT-Py	$\frac{\text{DYN-s}}{\text{DYN-c}}$	$\frac{\text{STAT-SQL}}{\text{DYN-c}}$	$\frac{\text{STAT-Py}}{\text{DYN-c}}$
0.05	10000	12259.6	1124	28	-5.8	0.25	0.56	1.07	0.3	2.21	4.24	1.19
0.05	20000	24577.6	2229.8	32	-6	0.4	1.06	1.83	0.56	2.67	4.6	1.4
0.05	30000	36835.8	3377.2	21	-5.6	0.54	1.42	2.66	0.77	2.64	4.95	1.43
0.05	40000	48991.6	4530.2	28	-6	0.73	1.94	3.55	0.98	2.65	4.86	1.34
0.05	50000	61259	5649.8	27	-5.6	0.84	2.21	4.39	1.22	2.65	5.25	1.45
0.05	75000	91862	8457.6	31	-6	1.26	3.45	6.12	1.84	2.75	4.87	1.47
0.05	100000	122644	11291	20	-5.8	1.59	4.39	8.25	2.34	2.76	5.19	1.47
0.1	10000	24546.4	261.4	34	-6	0.22	0.5	1.53	0.29	2.27	6.92	1.31
0.1	20000	48911	519.4	49	-6	0.33	0.88	2.71	0.55	2.67	8.19	1.66
0.1	30000	73412.4	789.8	43	-6	0.46	1.16	3.84	0.78	2.53	8.34	1.69
0.1	40000	97919.4	1034.6	56	-6	0.57	1.41	4.86	0.99	2.48	8.52	1.74
0.1	50000	122317	1297.6	47	-6	0.64	1.69	5.81	1.23	2.64	9.05	1.92
0.1	75000	183908.8	1923.6	47	-6	0.83	2.56	8.68	1.84	3.08	10.43	2.21
0.1	100000	245126.4	2575.8	48	-6	1.13	3.39	11.66	2.52	3	10.33	2.23
0.15	10000	36659.4	204.2	72	-6	0.21	0.44	1.9	0.32	2.09	8.92	1.51
0.15	20000	73491.2	406	71	-6	0.30	0.68	3.14	0.56	2.24	10.38	1.85
0.15	30000	110142.8	609.8	70	-6	0.35	0.87	4.57	0.78	2.49	13.05	2.23
0.15	40000	147038	815.4	73	-6	0.42	1.14	5.83	1.01	2.72	13.88	2.4
0.15	50000	183608.2	1017.6	59	-6	0.48	1.45	7.04	1.24	3.01	14.61	2.58
0.15	75000	275649.8	1527.8	62	-6	0.73	2.25	10.74	1.86	3.07	14.67	2.55
0.15	100000	367722.8	2037.6	65	-6	0.98	2.95	14.4	2.57	3.02	14.76	2.63
0.2	10000	48912	200.2	99	-6	0.19	0.42	2.19	0.31	2.21	11.65	1.63
0.2	20000	97934.6	400.6	94	-6	0.23	0.54	3.44	0.56	2.35	14.93	2.42
0.2	30000	146978.6	600.6	108	-6	0.29	0.79	5.34	0.8	2.72	18.36	2.75
0.2	40000	195696.4	801	98	-6	0.38	1.05	6.79	1.01	2.79	18.07	2.68
0.2	50000	245184.2	1000.8	89	-6	0.43	1.33	8.56	1.28	3.11	20	2.99
0.2	75000	367597.4	1501.2	91	-6	0.68	2.08	12.84	1.87	3.05	18.87	2.74
0.2	100000	490189.4	2002	107	-6	0.9	2.77	17.19	2.65	3.09	19.19	2.96
0.25	10000	61323.2	200	124	-6	0.20	0.39	2.51	0.31	1.92	12.4	1.55
0.25	20000	122483.6	400	107	-6	0.22	0.5	4.06	0.57	2.21	18.11	2.53
0.25	30000	183640.2	600	127	-6	0.28	0.76	6.06	0.8	2.7	21.46	2.87
0.25	40000	244909.2	800	119	-6	0.35	1	8.23	1.03	2.82	23.22	2.9
0.25	50000	306185	1000	116	-6	0.41	1.26	10.08	1.29	3.08	24.6	3.15
0.25	75000	459241.8	1500.4	134	-6	0.64	1.98	14.94	1.92	3.08	23.18	2.98
0.25	100000	612608	2000	104	-6	0.86	2.65	20.17	2.69	3.07	23.33	3.12
0.3	10000	73516.6	200	137	-6	0.16	0.28	2.56	0.3	1.8	16.36	1.93
0.3	20000	146868.2	400	138	-6	0.2	0.49	4.6	0.57	2.42	22.62	2.82
0.3	30000	220461.2	600	135	-6	0.28	0.75	6.96	0.81	2.68	24.9	2.89
0.3	40000	294246.4	800	159	-6	0.35	0.97	9.02	1.11	2.76	25.64	3.16
0.3	50000	367958.6	1000	160	-6	0.4	1.22	11.11	1.3	3.05	27.76	3.25
0.3	75000	551294.2	1500	115	-6	0.63	1.93	16.84	1.92	3.06	26.74	3.05
0.3	100000	735921.8	2000	142	-6	0.83	2.56	22.59	2.72	3.1	27.35	3.29

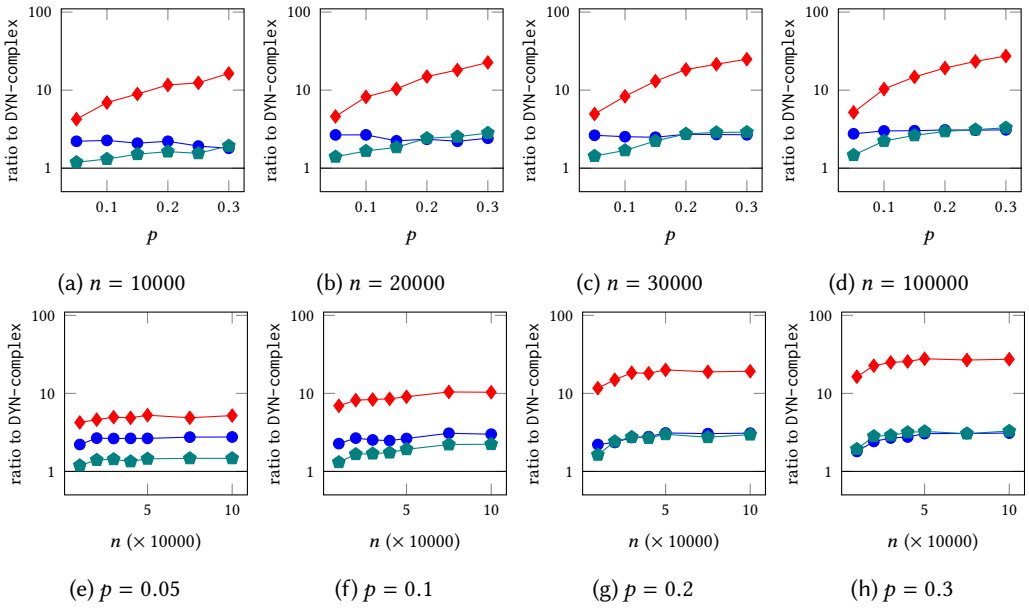

Fig. 2. Ratios of the running times in Experiment 2 of the different evaluation methods compared to DYN-complex, for different values of n and p, as listed in Table 2. ●: $\frac{\text{DYN-single}}{\text{DYN-complex}}$ ♦: $\frac{\text{STATIC-SQL}}{\text{DYN-complex}}$ ⬟: $\frac{\text{STATIC-Python}}{\text{DYN-complex}}$

Table 3. Results of the third experiment (application of the changes of the first two experiments to a large real-world graph). See Table 1 for an explanation of the columns. Timings for STATIC-SQL are not listed, as the experiment did not finish within twelve hours.

Graph	$\lvert V\rvert$	$\lvert E\rvert$	cc's	change	$\Delta\lvert E\rvert$	Δcc's	DYN-c	DYN-s	STAT-SQL	STAT-Py	$\frac{\text{DYN-s}}{\text{DYN-c}}$	$\frac{\text{STAT-SQL}}{\text{DYN-c}}$	$\frac{\text{STAT-Py}}{\text{DYN-c}}$
G_{dblp}	1949121	8823792	151874	ρ_1	1950	-248	51.29	6270.86	–	59.65	122.26	–	1.16
G_{dblp}	1949121	8823792	151874	ρ_2	140	-3	42.1	38.11	–	52.31	0.91	–	1.24

time of STATIC-SQL and STATIC-Python on the other hand is dominated by the size of the graph, and for STATIC-SQL in particular by the size and the density of the connected components. So, the relative performances of the different evaluation methods very much depend on the test case at hand.

6 BOUNDED BRIDGE DISTANCE

In this section, we prove the bridge bounds that were stated in Section 4.

We first show that every first-order definable insertion query has the undirected bounded bridge property. The proof makes use of the result by Feferman-Vaught that the rank-k first-order type of the disjoint union of two structures is determined by the rank-k first-order types of these two structures [19, 20] (see also [31]).

The *rank-k type* of a tuple $\bar{a}$ of arity ℓ with respect to a structure $\mathcal{S}$ is the set of all first-order formulas $\varphi(\bar{x})$ of quantifier-rank k with ℓ free variables such that $\mathcal{S} \models \varphi(\bar{a})$ (cf. [29, Definition 3.14]). Here the quantifier-rank of a formula is its maximum nesting depth of quantifiers. We denote the set of rank-k types of tuples of arity ℓ by FO$[k, \ell]$. Note that because there are only finitely

many semantically non-equivalent first-order formulas of quantifier-rank k with ℓ free variables, the set FO$[k, \ell]$ is finite for all k and ℓ, although of non-elementary size.

Our proof of Proposition 4.1 uses the following version of the Theorem of Feferman and Vaught.

Theorem 6.1 (Theorem of Feferman and Vaught [19, 20]). *Let $\mathcal{A}$ and $\mathcal{B}$ be two structures. Then the rank-k types of a tuple $\bar{a}$ of $\mathcal{A}$ and a tuple $\bar{b}$ of $\mathcal{B}$ uniquely determine the rank-k type of the tuple $(\bar{a}, \bar{b})$ in the disjoint union of $\mathcal{A}$ and $\mathcal{B}$.*

Proof (of Proposition 4.1). We show that each first-order definable insertion operation ρ with underlying first-order formula $\mu(\bar{p}; \bar{x})$ of quantifier-rank k has the undirected bounded bridge property. Let ℓ be the arity of $\bar{p}$ and c' the number of FO$[k, 1]$-types of directed graphs. We will bound the number of bridges on undirected paths created by ρ by $c \stackrel{\text{def}}{=} \ell + c'$.

To this end let G be a graph and let $\delta = \rho(\bar{a})$ for some tuple $\bar{a}$ of nodes of G. Let u, v be two nodes that are connected by some path π of the form $u = w_0, w_1, \ldots, w_r = v$ in $\delta(G)$. Let q be the number of bridges of π, that is, edges that are not in G. Our goal is to show that there exists such a path with at most c bridges. For $q \leq c$, there is nothing to prove, so we assume $q > c$. It suffices to show that there is a path from u to v with fewer than q bridges. Let $(u_1, v_1), \ldots, (u_q, v_q)$ be the bridges in π. If for some i, the nodes u_i and v_i are in the same weakly connected component of G (before the application of δ), we can replace the bridge (u_i, v_i) by a path of "old" edges resulting in an overall path with $q - 1$ bridges. Similarly, if u_i and u_j are in the same weakly connected component of G, for some $i < j$, we can shortcut π by a path from u_i to u_j inside G. Therefore, we can assume that, for every i, the nodes u_i and v_i are in different weakly connected components of G, and likewise u_i and u_j for $i < j$.

We show that in this case there are i, j with $i < j$ such that μ defines an edge between u_i and v_j, and therefore a path with fewer bridges can be constructed by shortcutting the path π with the edge (u_i, v_j). Indeed, by the choice of m there must be two nodes u_i and u_j, with $i < j$, in distinct weakly connected components of G that do not contain any element from $\bar{a}$, such that u_i and u_j have the same FO$[k, 1]$-type in their respective connected components. By Theorem 6.1, it follows that $(u_i, v_j, \bar{a})$ and $(u_j, v_j, \bar{a})$ have the same FO$[k, \ell + 2]$-types and therefore, since μ defines an edge between u_j and v_j, it also defines an edge between u_i and v_j. Clearly, the path $u, \ldots, u_i, v_j, \ldots, v$ has fewer bridges than π. □

We next turn to the undirected bridge bounds for UCQ-defined insertion queries.

Proof (of Proposition 4.3). Before we prove the three statements, we introduce some further notation. An insertion formula $\varphi(\bar{p}; x, y)$ may have a free tuple $\bar{p}$ of variables for parameters and two variables x, y that represent the two nodes of edges to be inserted. We write $(G, \bar{a}, u, v) \models \varphi$ to indicate that φ becomes true in G under the valuation η, which maps $\bar{p}$ to $\bar{a}$, x to u and y to v. For a UCQ $\varphi = \bigvee_{i=1}^{\ell} \exists \bar{z} \psi_i$ this holds if and only if $(G, \bar{a}, u, v) \models \exists \bar{z} \psi_i$, for some i. For any disjunct CQ ψ_i it holds $(G, \bar{a}, u, v) \models \psi_i$ if η can be extended to a valuation that also maps the variables from $\bar{z}$ such that all atomic formulas in ψ_i become true.[18] In the following, we associate a (hyper-)graph with a conjunctive query as it is common in the literature on CQs [22]. For each CQ ψ_i we consider its set M_i of maximal weakly connected components in the common (hyper-)graph associated with ψ_i. We note that each weakly connected component of ψ_i can be mapped to a different weakly connected component of the graph. Indeed, e ach component in M_i yields a CQ $\theta = \exists \bar{y} \theta'$, where θ' is the conjunction of all atoms of the component and $\bar{y}$ contains only the variables of the component. We call these formulas the *patterns* of ψ_i. For each i, we denote the pattern in which x occurs by

[18] We use the standard semantics for UCQs and CQs here, but describe it in a way that facilitates our arguments below.

θ_i^x and likewise for θ_i^y. We note that $(G, \bar{a}, u, v) \models \psi_i$ holds if and only if $(G, \bar{a}, u, v) \models \theta$, for every pattern $\theta \in M_i$.

In the following, we assume, without loss of generality, that θ_i^x and θ_i^y are always different: if x and y are in the same connected component of ψ_i, then the pattern $\theta_i^x = \theta_i^y$ can only be satisfied by valuations that map x and y to nodes that are connected by an undirected path, and thus new edges are only inserted inside (weakly) connected components of a graph. We write $(G, \bar{a}, x/u) \models \theta_i^x$ to indicate that θ_i^x becomes true in G under an valuation which maps $\bar{p}$ to $\bar{a}$ and x to u. Similarly, $(G, \bar{a}, y/v) \models \theta_i^y$.

For the proof of the upper bound of Statement (a), let us assume, towards a contradiction, that there is a graph G, a change $\delta = \rho(\bar{a})$ and two nodes u, v such that $\mathrm{ubd}_{G,\delta(G)}(u, v) \geq 2\ell + 1$, that is, there is an undirected path π in $\delta(G)$ from u to v with at least $2\ell + 1$ bridges, and there is no such path with fewer bridges. By the pigeonhole principle, at least three (not necessarily consecutive) bridges $(u_1, v_1), (u_2, v_2), (u_3, v_3)$ from π are inserted by the same CQ ψ_i. In particular, $(G, \bar{a}, u_1, v_1) \models \psi_i$ and $(G, \bar{a}, u_3, v_3) \models \psi_i$. Without loss of generality, we assume that u_1 is the first of these six nodes that occurs on the path from u to v and that (u_3, v_3) is the last of the three edges on this path.[19] In particular, neither u_3 nor v_3 can be in the weakly connected component of v_1 in G.

From $(G, \bar{a}, u_1, v_1) \models \psi_i$ and $(G, \bar{a}, u_3, v_3) \models \psi_i$ we can conclude that $(G, \bar{a}, x/u_1) \models \theta_i^x$ and $(G, \bar{a}, y/v_3) \models \theta_i^y$. For all other patterns θ of ψ_i it holds $(G, \bar{a}) \models \theta$. Altogether this yields $(G, \bar{a}, u_1, v_3) \models \psi_i$ and thus δ also inserts the edge (u_1, v_3). However, this edge can be used to shortcut π, resulting in a path from u to v with fewer bridges, the desired contradiction, thus showing Statement (a).

We now show that the bound is actually tight. For any prime q let $\chi_q(z)$ be the conjunctive query $\exists z_1, \ldots, z_{q-1} E(z, z_1) \wedge E(z_1, z_2) \wedge \cdots \wedge E(z_{q-1}, z)$ expressing that z lies on a cycle of length q. For each $i \leq \ell$, let $\psi_i(x, y) = \chi_{q_{2i-1}}(x) \wedge \chi_{q_{2i}}(y)$, where q_j denotes the jth prime number. That is, an edge (u, v) is inserted due to ψ_i whenever u is part of a cycle of length q_{2i-1} and v is part of a cycle of length q_{2i}, for some i. Furthermore, for every prime q, let C_q be a cycle graph with q nodes.

For any prime numbers p, q, it is easy to see that $(C_p, u) \models \chi_q(z)$ holds for a node u of a cycle C_p if and only if $p = q$.

To show that for CQs the bridge bound 2 is sharp, we can consider the insertion formula $\psi_1(x, y) = \chi_2(x) \wedge \chi_3(y)$ and let G be a graph consisting of three disjoint cycles, two of length 2, and one of length 3. Let u, v be two nodes from the two different length 2 cycles. It is easy to see that ψ_1 introduces an edge from every 2-cycle node to every 3-cycle node and thus produces an undirected path with 2 bridges from u and v, but there is no such path with fewer bridges.

For arbitrary $\ell > 1$ we need a slightly more complicated construction. To this end, let $\varphi(x, y) = \bigvee_{i=1}^{\ell} \exists \bar{z} \psi_i(x, y)$ and, for numbers m_1, m_2, let D_{m_1, m_2} be a graph consisting of a cycle of length m_1 and a cycle of length m_2, which have one node in common. Finally, let G be the disjoint union of

- $\ell + 2$ cycles $C_{q_1}, C_{q_2}, C_{q_4} \ldots, C_{q_{2\ell-2}}, C_{q_{2\ell-1}}, C_{q_{2\ell}}$, and
- $\ell - 1$ graphs $D_{q_1, q_3}, D_{q_3, q_5}, \ldots, D_{q_{2\ell-3}, q_{2\ell-1}}$.

Now it is not hard to verify that, if u is some node from C_{q_1} and v from $C_{q_{2\ell-1}}$ the application of φ yields paths from u to v through the subgraphs $C_{q_2}, D_{q_1, q_3}, C_3, D_{q_3, q_5}, C_5, \ldots, D_{q_{2\ell-3}, q_{2\ell-1}}, C_{q_{2\ell}}$, in that order (see Figure 3). Each of these paths has 2ℓ bridges and it is not hard to see that there is no path with fewer bridges since edges are only inserted between consecutive subgraphs in the above sequence.

[19] However, since we consider undirected paths, we do not know whether u_3 is visited before v_3 in π.

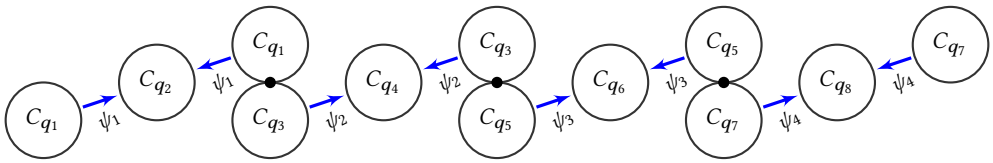

Fig. 3. Illustration of the graph from the proof of Proposition 4.1a for $\ell = 4$. Each C_{q_i} depicts a cycle of length q_i. Bridges are highlighted in blue and annotated by the conjunct used for their insertion.

For a proof of Statement (b) we only need to extend the argument for the upper bound of Statement (a) slightly. Conceptually, we view a CQ with negation as a formula of the form $\exists \bar{z} \psi \wedge \psi'$, where ψ is a conjunction of positive atoms and ψ' a conjunction of negated atoms. We recall that each variable of a CQ with negation needs to occur in some positive atom. For such a query to hold in a database, the CQ $\exists \bar{z} \psi$ has to become true in the same way as above, by making all its patterns true. We refer to these patterns in the following as the *positive patterns*. Furthermore, for some valuation which witnesses that $\exists \bar{z} \psi$ holds, all negated atoms of ψ' need to hold. Since E is the only available relation symbol, this rules out the existence of some edges between the nodes in the range of the valuation.

We will show a slightly stronger upper bound than stated in the proposition.[20] Let $\rho(\bar{p})$ be a UCQ$^\neg$ $\varphi(\bar{p}; x, y) = \bigvee_{i=1}^{\ell} \exists \bar{z} \psi_i(\bar{p}; x, y)$ where each ψ_i has at most k positive patterns. We show that the undirected bridge bound of ρ is at most $2(k-1)\ell$.

Towards a contradiction, we now assume that there is a graph G, a change $\delta = \rho(\bar{a})$ and two nodes u, v such that $\mathrm{ubd}_{G,\delta(G)}(u, v) \geq 2(k-1)\ell + 1$, as witnessed by an undirected path π in $\delta(G)$ from u to v with at least $2(k-1)\ell + 1$ bridges, and the absence of a path with fewer bridges. By the pigeonhole principle, there is a CQ$^\neg$ ψ_i that inserts at least $2k-1$ (not necessarily consecutive) bridges $(u_1, v_1), \ldots, (u_{2k-1}, v_{2k-1})$ of π. Let again, u_1 be the first of these nodes in π and let w be the last (i.e., $w = u_{2k-1}$ or $w = v_{2k-1}$). We fix a valuation η which witnesses $(G, \bar{a}, u_1, v_1) \models \psi_i$. To make the $k-2$ positive patterns of ψ_i, which do not contain x or y, true, variables of $\bar{z}$ can be mapped into at most $k-2$ weakly connected components of G. If some weakly connected component of G would contain more than two of the nodes $v_3, \ldots, v_{2k-1}$ under η, π could be shortcut yielding a path from u to v with fewer bridges. Likewise, none of them can be from the components of u_1 or v_1. Therefore, at least one of the nodes $v_3, \ldots, v_{2k-1}$ must be from a component of G that is not in the range of η. Let j be chosen, such that v_j is such a node and let η' be a valuation that witnesses $(G, \bar{a}, u_j, v_j) \models \psi_i$. Since η' maps the variables of θ_i^y into a component of G that is not in the range of η, the valuation η'' that coincides with η' on the variables of θ_i^y and with η everywhere else witnesses $(G, \bar{a}, u_i, v_j) \models \psi_i$. Therefore, there is a shortcut edge from u_1 to v_j, the desired contradiction. It is worth mentioning here, that we can be sure that for all negated atoms of the form $\neg E(z_1, z_2)$ where z_1 is from θ_i^y and z_2 from some other positive pattern of ψ_i, there is no edge from $\eta''(z_1)$ to $\eta''(z_2)$, since these nodes are in different components of G (and likewise for atoms $\neg E(z_2, z_1)$). This completes the proof of Statement (b).

Towards (c), our goal is to construct a CQ$^\neg$-defined insertion query ρ as well as a graph G consisting of connected components $D_0, \ldots, D_n$, each D_i with a distinguished node v_i such that the application of ρ inserts exactly the edges $(v_0, v_1), \ldots (v_{n-1}, v_n)$. Then the undirected bridge distance between v_0 and v_n will be n.

[20] We opted for a weaker Statement (b), since we did not want to define the notion of pattern before stating the proposition.

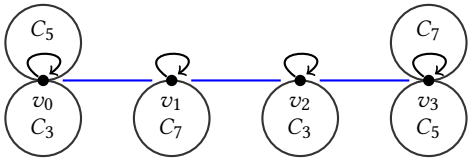

Fig. 4. Illustration of the graph from the proof of Proposition 4.3c for $n = 3$ with $Q = \{3, 5, 7\}$, $q(0, 2) = 3$, $q(0, 3) = 5$, and $q(1, 3) = 7$. Each C_i depicts a cycle of length i and bridges are highlighted in blue. As an example, the cycles of length three prevent the insertion of the edge (v_0, v_2).

The distinctive feature of the nodes v_i will be self-loops. The remaining challenge is to construct the insertion query and the components of G such that no edge between components D_i and D_j is inserted if $|i-j| > 1$. To this end, let, for each $i, j \in \{0, \ldots, n\}$ with $|i-j| > 1$, $q(i, j)$ be a distinct prime number and let Q be the set of all these prime numbers, that is, $Q = \{q(i, j) \mid 0 \leq i, j \leq n, |i-j| > 1\}$[21]. For each $i \in \{0, \ldots, n\}$, let Q_i be the set $\{q(i, j) \mid 0 \leq j \leq, |i - j| > 1\}$. For a finite set N of numbers let D_N be a graph with one distinguished node v with a self-loop, and one directed cycle of length q, for each $q \in N$, all of them containing the node v and otherwise being disjoint. Finally, the graph G is the disjoint union of graphs $D_{Q_0}, \ldots, D_{Q_n}$, where in each D_{Q_i} the distinguished node is denoted by v_i (see Figure 4).

Now we construct a $\mathrm{CQ}^\neg$ insertion formula that inserts exactly the edges $(v_0, v_1), \ldots, (v_{n-1}, v_n)$ into G. To this end, it inserts an edge (u, v) if (1) u, v have self-loops, (2) for each $q \in Q$, there is some cycle of length q in G, which avoids u and v. For each k, let

$$\chi_k(x, y, x_1, \ldots, x_k) = E(x_1, x_k) \wedge \bigwedge_{i=1}^{k-1} E(x_i, x_{i+1}) \wedge \bigwedge_{i=1}^{k} (\neg E(x, x_i) \wedge \neg E(y, x_i)).$$

That is, χ_k expresses that the variables $x_1, \ldots, x_k$ are mapped to a closed path of length k and that x and y do not occur on that cycle. Finally, the insertion query ρ is induced by the $\mathrm{CQ}^\neg$ formula

$$\varphi(x, y) = E(x, x) \wedge E(y, y) \wedge \bigwedge_{q \in Q} \exists x_1, \ldots, x_q \ \chi_q(x, y, x_1, \ldots, x_q).$$

We claim that the undirected bridge distance between v_0 and v_n in G is n with respect to the insertion ρ. It is at most n since the edges $(v_0, v), \ldots (v_{n-1}, v_n)$ are inserted by ρ. On the other hand, no edge between components D_i and D_j is inserted if $|i - j| > 1$. To see this, we observe first that φ only allows to insert edges between distinguished nodes v_i and v_j due to the required self-loop. Towards a contradiction, let us assume that ρ inserts an between nodes v_i and v_j with $|i - j| > 1$. Let $q = q(i, j)$. However, cycles of length q only occur in D_i and D_j and can thus not be used to satisfy $\exists x_1, \ldots, x_q \ \chi_q(x, y, x_1, \ldots, x_q)$, the desired contradiction. □

Finally, we give the proof for the bridge bound for quantifier-free insertion queries on directed acyclic graphs.

Proof (of Proposition 4.4). We show that each quantifier-free insertion operation ρ with underlying formula $\mu(\bar{p}; \bar{x})$ has the bounded bridge property on acyclic graphs. Let ℓ be the arity of the parameter tuple of ρ and c' the number of FO$[0, \ell + 1]$-types of graphs. We will bound the number of bridges on paths created by ρ by $c \stackrel{\text{def}}{=} c' + 1$.

[21] Thanks to the ability to use negated atoms it is not strictly necessary to work with primes. However, we aim at a construction that indicates where negated atoms are really *needed*.

To this end let G be a directed, acyclic graph, let $\delta = \rho(\bar{a})$ be a change, and let u, v be nodes of G. As in the proof of Proposition 4.1, we show that each path π from u to v with $q > c'$ bridges can be transformed into a path with fewer bridges, unless a cycle with at most c bridges is introduced.

To this end, let $(u_1, v_1), \ldots, (u_q, v_q)$ be the bridges in π. Since q is larger than the number c' of FO$[0, \ell + 1]$-types, there are i, j with $1 \leq i < j \leq c'$ such that $(v_i, \bar{a})$ and $(v_j, \bar{a})$ have the same FO$[0, \ell + 1]$-types. We distinguish three cases. In case (1), the edge (v_i, u_i) is in G and thus δ introduces a cycle of length 2. In case (2), the edge (v_j, u_i) is in G and, together with the sub-path from u_i to v_j, constitutes a cycle with at most c bridges. In case (3), u_i is neither connected to v_i nor to v_j by an edge. Therefore $(u_i, v_i, \bar{a})$ and $(u_i, v_j, \bar{a})$ have the same FO$[0, \ell + 2]$-types and δ inserts an edge (u_i, v_j) as well, the desired shortcut. □

7 INEXPRESSIBILITY RESULTS

So far we have seen examples of queries that can be maintained by DynFO programs under definable changes. While, e.g., reachability can be maintained under definable insertions for restricted graphs, our current techniques seem not to suffice for maintaining reachability under definable insertions *and* deletions, even for restricted graph classes. In this section we approach the question whether there are inherent obstacles towards such results, that is, we aim at proving inexpressibility results for reachability.

It is notoriously difficult to show that a query *cannot* be maintained by a DynFO program. Indeed, for single-tuple changes there are no inexpressibility results for DynFO besides those that follow from the easy observation that every query that can be maintained in DynFO is computable in polynomial time (and thus, e.g., problems complete for exponential time are not in DynFO). One reason is that standard methods for proving inexpressibility for first-order logic with arithmetic cannot be applied to DynFO. In particular, typical examples of queries that are not expressible in first-order logic, such as the reachability query or the query asking for the parity of a unary relation, can be maintained in DynFO under single-tuple changes.

Previous work on inexpressibility for single-tuple changes therefore focused on the search for good techniques, mainly by studying restrictions of DynFO. Most notably, lower bound methods have been developed for fragments obtained by restricting the use of auxiliary relations, either structurally or by arity (see for example [10, 14, 15]), and the fragment where update rules are quantifier-free first-order formulas (see for example [21, 45, 47]).

While it should be easier to prove inexpressibility results for DynFO in the presence of first-order definable change operations, it is highly unlikely, as for single-tuple changes, that standard methods can be adopted easily. Therefore we follow the same approach and start by studying restrictions of DynFO. We confirm that, unsurprisingly, complex change operations can make it harder to maintain a query. Even more, our results show that dynamic programs for complex changes may not exist, even if they do for single-edge changes.

Towards our first set of results, presented in Subsection 7.1, we recall that the reachability query *can* be maintained under single-tuple insertions with the transitive closure of the edge relation as only auxiliary relation (see Example 3.2). We show that this is not possible in the presence of a very simple complex insertion which is quantifier- and parameter-free, even if one allows additional unary auxiliary relations. This complements the result in [15] that unary auxiliary relations (besides the transitive closure itself) do not suffice for single-tuple deletions.

For acyclic graphs, the reachability query can be maintained under single-edge insertions *and* deletions with transitive closure as only auxiliary relation [13, 34]. As we have seen, this even holds in the presence of quantifier-free insertions (Theorem 4.5). We complement these results by showing that this approach does neither generalise to insertions with quantifiers nor to complex deletions. More precisely, we show that there is (1) an insertion query that uses only one quantifier

and (2) a quantifier- and parameter-free deletion query for each of which the reachability query cannot be maintained with the transitive closure and unary relations as auxiliary relations.

In a second line of work we study DynProp, the restriction of DynFO to quantifier-free update formulas. The dynamic program provided in Example 3.2 shows that quantifier-free update formulas suffice to maintain reachability under single-edge insertions. In Subsection 7.2 we prove that q_{Reach} cannot be maintained in DynProp under definable, quantifier-free insertions. We note that for single-edge deletions, despite some partial inexpressibility in this direction [47], it is still unknown whether Reachability can be maintained by quantifier-free update formulas.

We remark again that all inexpressibility results immediately transfer to the FOIES framework.

7.1 Inexpressibility of Reachability with Restricted Auxiliary Relations

We show that, in the presence of complex changes, the transitive closure, i.e., the query relation, as the only binary auxiliary relation does not suffice to maintain reachability. For this result to hold, it suffices to allow single-tuple insertions and one complex change query, which can be chosen either as an insertion or a deletion query. This change query can be restricted in several ways and in most cases the result even holds for acyclic graphs. These results should be contrasted with Theorem 4.5. By Σ_1 we denote the existential fragment of first-order logic, that is, the set of first-order formulas in prenex normal form with only existential quantification.

Theorem 7.1. *(a) There is an insertion query ρ_1 such that $(q_{Reach}, \{\text{INS}_E, \rho_1\})$ cannot be maintained in DynFO, if all auxiliary relations besides the query relation are unary. The insertion query ρ_1 can be chosen as quantifier-free and parameter-free.*
The result even holds on acyclic graphs, in which case ρ_1 can be chosen as Σ_1-definable and parameter-free.

(b) There is a deletion query ρ_2 such that $(q_{Reach}, \{\text{INS}_E, \rho_2\})$ cannot be maintained in DynFO, if all auxiliary relations besides the query relation are unary. The deletion query ρ_2 can be chosen as quantifier-free and parameter-free.
The result also holds on acyclic graphs, even with a quantifier-free ρ_2.

Theorem 7.1 (a) even holds in the case where the nodes of G are linearly ordered but the proof becomes considerably more involved. It can be found in [37, Theorem 10].[22]

The proof of Theorem 7.1 makes use of suitable static lower bounds. This approach has been used often before and was made precise in [44].

We say that a k-ary query q is expressed by a formula $\varphi(\bar{x})$ with *help relations of schema* τ, if, for every database $\mathcal{D}$, there is a τ-database H over the same domain such that for every k-tuple $\bar{a}$ it holds: $\bar{a} \in q(\mathcal{D})$ if and only if $(\mathcal{D}, H) \models \varphi(\bar{a})$. This notion should not be confused with definability of the query q in existential second-order logic. In the latter case, the relations can be chosen depending on $\bar{a}$, but here the relations need to "work" for all tuples $\bar{a}$.

The proof of Theorem 7.1 relies on the fact that unary help relations do not suffice to express the transitive closure for path graphs, i.e., graphs consisting of just one path, and for unions of paths even if a certain special binary relation is present. Before stating this formally, we define the latter class more precisely. For $n, m \in \mathbb{N}$, let $G_{n,m} = (V_{n,m}, E_{n,m}, \text{pos})$ where $(V_{n,m}, E_{n,m})$ is the disjoint union of n paths of m nodes each, that is, $V_{n,m} = \{v_{i,j} \mid i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}\}$, and $E_{n,m} = \{(v_{i,j}, v_{i,j+1}) \mid i \in \{1, \ldots, n\}, j \in \{1, \ldots, m-1\}\}$. The binary relation pos contains tuples $(v_{i,j}, v_{i',j'})$ with $j < j'$, that is, $v_{i,j}$ is at a smaller position than $v_{i',j'}$ in its path. We denote the class of all databases of the form $G_{n,m}$, for arbitrary $n, m \in \mathbb{N}$, by $\mathcal{G}_{\text{up}}$.

[22] We strongly conjecture that Theorem 7.1 (b) holds in the case with a linear order as well.

Lemma 7.2. *The query q_{Reach} cannot be expressed by a first-order formula with unary help relations on (a) path graphs and (b) databases from $\mathcal{G}_{up}$.*

Statement (a) follows from an easy locality-based argument and was proved in [44, Lemma 4.3.2]. Statement (b) follows from a standard Ehrenfeucht-Fraïssé argument.

Proof (of Theorem 7.1). In order to prove (a), let us assume, towards a contradiction, that for every quantifier-free and parameter-free insertion query ρ_1, $(q_{\text{Reach}}, \{\textit{ins}_E, \rho_1\})$ can be maintained in DynFO on directed graphs with k unary auxiliary relations $B_1, \ldots, B_k$.

Our goal is to show that then the transitive closure of path graphs *could* be expressed by a first-order formula with unary help relations $B_1, \ldots, B_k$ and C_0, C_1, C_2.

We refer to Figure 5 for an illustration of the following. Let G be some arbitrary path graph. Let C_0, C_1, C_2 be unary relations as indicated in Figure 5: the relation C_i contains all nodes whose position in the path is i modulo 3. Let G_1 be derived from G by reversing all edges from C_1-nodes to C_2-nodes and by adding loops to all C_1-nodes and C_2-nodes. Let G_2 be the result of applying the insertion query $\rho_1 = \mu_E(x,y) \stackrel{\text{def}}{=} E(x,y) \vee \Big(E(x,x) \wedge E(y,y) \wedge E(y,x)\Big)$ to G_1. Intuitively, ρ_1 adds all edges (x,y) for which there is an edge (y,x) and both x and y have self-loops.

By our assumption, $q_{\text{Reach}}(G_2)$ can be defined by a first-order formula ψ_2 using $q_{\text{Reach}}(G_1)$ and unary auxiliary relations $B_1, \ldots, B_k$. We next show why this implies that the transitive closure of path graphs G can be expressed by a first-order formula with $B_1, \ldots, B_k$ and C_0, C_1, C_2 as unary help relations.

First, there is a simple formula φ_1 which defines the edge relation E_1 of G_1 in terms of the edge relation E of G and C_0, C_1, C_2. Since all directed paths in G_1 have length at most 2, $q_{\text{Reach}}(G_1)$ can be defined by a first-order formula ψ_1 on G_1.

By combining φ_1, ψ_1, ρ_1, and ψ_2 in a suitable way it is straightforward to construct a first-order formula $\theta(x,y)$ which defines $q_{\text{Reach}}(G_2)$ on $(G, C_0, C_1, C_2, B_1, \ldots, B_k)$. It is also straightforward to define $q_{\text{Reach}}(G)$ from $q_{\text{Reach}}(G_2)$ in a first-order fashion, when C_0, C_1, C_2 and $q_{\text{Reach}}(G_2)$ are given (basically ignore all pairs (u,v), for which there is an edge (v,u) in G).

Altogether, we can conclude that the transitive closure query can be defined on G with the help of unary help relations $C_0, C_1, C_2, B_1, \ldots, B_k$, the desired contradiction.

We observe that both graphs G_1 and G_2 in the above proof are not acyclic. Yet a slight modification of the construction yields acyclic graphs G_1' and G_2'. However, it uses existential quantifiers in the definition of the change operation. The graphs are also depicted in Figure 5. The graph G_2' is obtained by applying the operation $\rho_1' = \mu_E(x,y) \stackrel{\text{def}}{=} E(x,y) \vee \Big(\exists z\Big(E(x,z) \wedge E(y,z)\Big) \wedge \exists z' E(z',x)\Big)$ to G_1'. The proof is now analogous to (a) except that G_1' has to be first-order interpreted into the path graph G, as it uses a slightly larger domain. The rest of the argument for acyclic graphs is analogous.

For (b), a similar approach is used, this time starting from a database $G'' = G_{n,m}$ from $\mathcal{G}_{\text{up}}$. The construction is illustrated in Figure 6. It only involves acyclic graphs. Let G_1'' be a graph derived from G'' by adding $n(m-1)$ nodes $\{w_{i,j} \mid i \in \{1,\ldots,n\}, j \in \{1,\ldots,m-1\}\}$ that are used to connect the paths: for each $i, i' \in \{1,\ldots,n\}$ and each $j \in \{1,\ldots,m-1\}$, the edges $(v_{i,j}, w_{i',j})$ and $(w_{i',j}, v_{i,j+1})$ are added. Notice that in G_1'' there is a path from a node $v_{i,j}$ to a node $v_{i',j'}$ with $(i,j) \neq (i',j')$ if and only if $j < j'$. Additionally, we add a node u and edges $(w_{i,j}, u)$ for all $i \in \{1,\ldots,n\}, j \in \{1,\ldots,m-1\}$. This node will be used as the parameter for the deletion. Let G_2'' be the result of applying the deletion query $\rho_2(p) = \mu_E(p;x,y) \stackrel{\text{def}}{=} E(x,y) \wedge \neg E(x,p) \wedge \neg E(y,p) \wedge y \neq p$ with parameter u, which deletes all edges that are incident to a node that has an edge to u, as well as all edges to u.

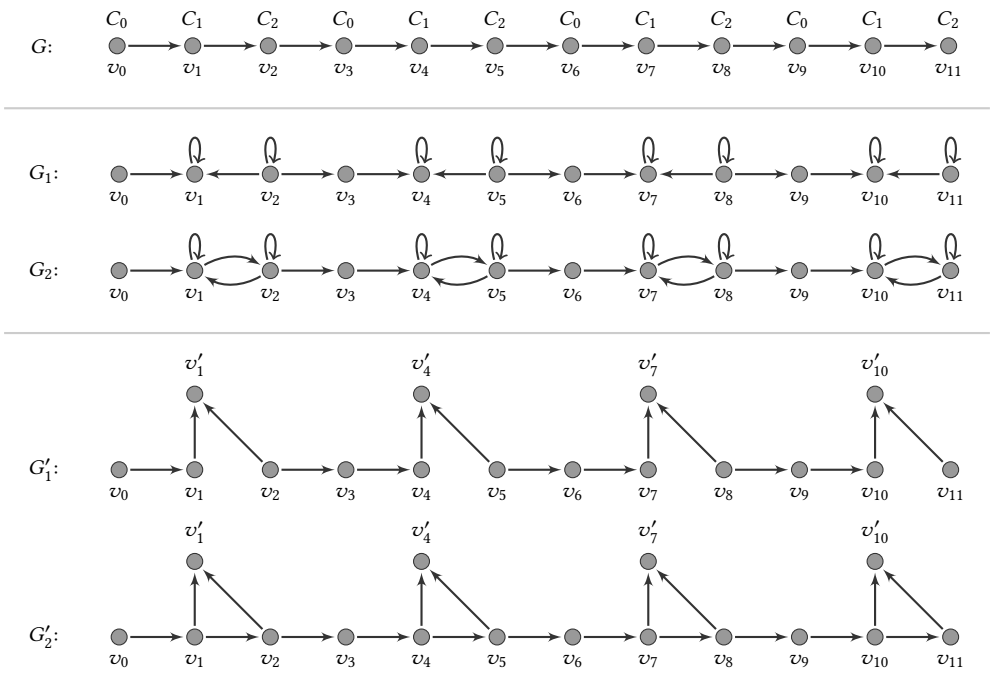

Fig. 5. The graphs from the proof of Theorem 7.1a.

The argument is now analogous to the previous proof. To this end we observe that the graph G_1'' can be first-order interpreted into G'' using pos[23]. Also, $q_{\text{Reach}}(G_1'')$ can be expressed by a first-order formula with the help of pos. Hence the combination of the interpretation formulas and an assumed formula for updating the transitive closure yields a formula expressing the transitive closure on $\mathcal{G}_{\text{up}}$. This proves the stated result for acyclic graphs.

In order to show that the result even holds for quantifier-free *and* parameter-free deletion queries on general graphs, the nodes $w_{i,j}$ can be equipped with self-loops, and the construction hence does need the parameter node u anymore.

□

7.2 Inexpressibility of Reachability in the Quantifier-Free Fragment

We now turn towards inexpressibility by quantifier-free update formulas. As discussed in the introduction of the section, studying this restriction is a first – though small – step towards finding methods for proving inexpressibility results for DynFO.

We conjecture that quantifier-free update formulas are too weak to maintain q_{Reach} even under single-tuple changes. Yet so far only restricted inexpressibility results have been obtained. Reachability cannot be maintained in DynProp under single-tuple changes when the auxiliary relations are at most binary or when the initialization is severely restricted [47]. For the more general alternating reachability query, quantifier-free update formulas do not suffice [21]. We next show that there are very simple quantifier-free change queries (a deletion or two insertions), with which q_{Reach} and even q_{UReach} cannot be maintained in DynProp.

[23]Because of this step, G_1'' contains $n(m-1)$ nodes of the form $w_{i,j}$. All other steps would also work using $m-1$ nodes of the form w_j.

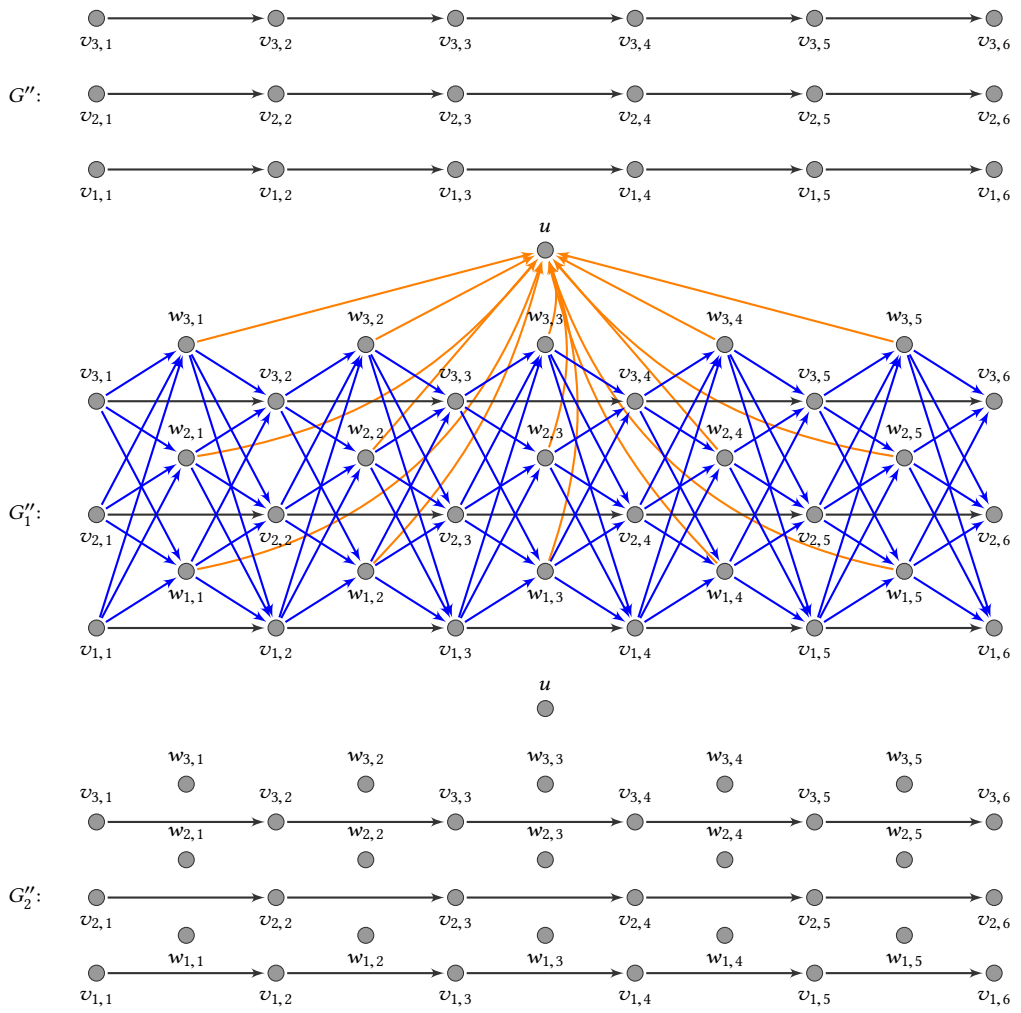

Fig. 6. The graphs from the proof of Theorem 7.1b (for $n = 3$ and $m = 6$)

Theorem 7.3. *(a) There is a quantifier-free deletion query ρ such that $(q_{Reach}, \{\text{INS}_E, \rho\})$ cannot be maintained in DynProp.*

(b) There are quantifier-free insertion queries ρ_1, ρ_2 such that $(q_{Reach}, \{\text{INS}_E, \rho_1, \rho_2\})$ cannot be maintained in DynProp.

The results also hold for acyclic and undirected graphs.

The following lemma introduces a tool for proving that some query is not in DynProp. It can be seen as a reformulation of the proof technique used in [43], which combines techniques from [21, 47] with insights regarding upper and lower bounds for Ramsey numbers. The lemma gives a sufficient condition under which a query q can not be maintained in k-ary DynProp under changes from a set Δ. This condition basically requires that for each collection S of subsets of size $k+1$ of a set $\{1, \ldots, n\}$ (for some large enough n), there is a database $\mathcal{D}$, a tuple $\bar{a}$ and a change sequence $\alpha(x_1, \ldots, x_{k+1})$ such that $\bar{a} \in q(\alpha(i_1, \ldots, i_{k+1})(\mathcal{D}))$ holds exactly if $\{i_1, \ldots, i_{k+1}\} \in S$. Here $\alpha(x_1, \ldots, x_{k+1})$ is a sequence of replacement queries with parameters from $\{x_1, \ldots, x_{k+1}\}$.

Furthermore, in $\mathcal{D}$ the elements $1, \dots, n$ can not be distinguished by quantifier-free formulas, even not with respect to the elements of $\bar{a}$. The actual statement of the lemma is slightly more general and allows α to use additional parameters.

In the following we denote the set $\{1, \dots, n\}$ by $[n]$ and the set of all subsets of size k of a set A by $\mathcal{P}_k(A)$. We write $(\mathcal{D}, \bar{a}) \equiv_0 (\mathcal{D}, \bar{b})$ if $\bar{a}$ and $\bar{b}$ are of the same arity and agree on their rank-0 type.

Lemma 7.4. *Let q be an m-ary τ_{in}-query and $\Delta \supseteq \Delta_{\tau_{in}}$ a set of quantifier-free replacement queries. Then (q, Δ) is not in k-ary DynProp if there is $r \in \mathbb{N}$ and an $n > n'$, for each $n' \in \mathbb{N}$, such that for all subsets $S \subseteq \mathcal{P}_{k+1}([n])$ there exist*

- *a τ_{in}-database $\mathcal{D}$ and tuples $\bar{a} = a_1, \dots, a_m$ and $\bar{u} = u_1, \dots, u_r$, such that*
  - *$[n]$ and $\{a_1, \dots, a_m, u_1, \dots, u_r\}$ are disjoint and contained in the domain of $\mathcal{D}$; and*
  - *$(\mathcal{D}, \bar{a}, \bar{i}, \bar{u}) \equiv_0 (\mathcal{D}, \bar{a}, \bar{j}, \bar{u})$ for all strictly increasing sequences $\bar{i}$ and $\bar{j}$ of length $k+1$ over $[n]$; and*
- *a sequence $\alpha(x_1, \dots, x_{k+1}, y_1, \dots, y_r)$ of changes where in each change at most one variable x_ℓ appears as parameter*

such that for all strictly increasing sequences $j_1, \dots, j_{k+1}$ over $[n]$ it holds that

$$\bar{a} \in q(\alpha(j_1, \dots, j_{k+1}, \bar{u})(\mathcal{D})) \quad \Longleftrightarrow \quad \{j_1, \dots, j_{k+1}\} \in S.$$

It should be noted that, although $\mathcal{D}$ can contain further elements, the parameters in the considered change sequences are only instantiated by elements from $[n] \cup \{u_1, \dots, u_r\}$. Of course, in the statement of this lemma, $[n]$ can be replaced by some other n-element set. There is even a more general form of it with tuples $\bar{t}_1, \dots, \bar{t}_n$ in place of $1, \dots, n$ and accordingly adapted conditions, but we decided to state this simpler form, since it suffices for Theorem 7.3.

We postpone the proof of Lemma 7.4 towards the end of this section and first use it to prove the desired lower bounds.

Proof (of Theorem 7.3). For both parts, the proof is by contradiction and uses Lemma 7.4.

For part (a) the idea is to start, for a given $S \subseteq \mathcal{P}_{k+1}([n])$, from a graph consisting of paths of length 2 of the form a_1, X, a_2, for every $X \in S$. The sequence α induced by a sequence $i_1, \dots, i_{k+1}$ deletes all edges (Y, a_2) for which some i_j is not in Y. It thus destroys all paths from a_1 to a_2, unless $\{i_1, \dots, i_{k+1}\} \in S$, just as required by Lemma 7.4; this yields the contradiction.

We make this more precise now. Let $\rho(p)$ be the quantifier-free deletion defined as $\rho(p) = \mu(p; x, y) \stackrel{\text{def}}{=} E(x, y) \wedge \neg E(p, x)$. Obviously, $\rho(p)$ deletes an edge (x, y) if there is an edge (p, x).

Towards a contradiction, let us assume that q_{Reach} can be maintained in k-ary DynProp under changes from $\{\textit{ins}_E, \rho\}$ for some k. We choose $m = 2$ and $r = 0$ in the statement of Lemma 7.4. Let $n \geq k + 1$ and $S \subseteq \mathcal{P}_{k+1}([n])$ be arbitrary.

The database $\mathcal{D}$ that we construct is actually a graph (see Figure 7 for an illustration). Its vertex set is of the form $[n] \cup \{a_1, a_2\} \cup S$, for elements a_1, a_2, such that $[n]$, $\{a_1, a_2\}$ and S are pairwise disjoint.

The graph has the following edges:

- For each $X \in S$ there are edges (a_1, X) and (X, a_2).
- For each $i \in [n]$ and for each $X \in S$ there is an edge (i, X) if $i \notin X$.

Since there are no edges between vertices from $\{a_1, a_2, 1, \dots, n\}$, the requirements of the lemma are fulfilled.

Intuitively, the nodes in $[n]$ control how edges from S to a_2 can be removed. To make this more precise, let us consider a sequence $1 \leq i_1 < \dots < i_{k+1} \leq n$ and let $Y \stackrel{\text{def}}{=} \{i_1, \dots, i_{k+1}\}$. The change sequence $\alpha = (\rho(i_1), \dots, \rho(i_{k+1}))$ removes all edges (X, a_2) with $i_j \notin X$, for some $j \leq k + 1$. Therefore, the only edge of the form (X, a_2) that might remain after applying α is (Y, a_2) which only exists if $Y \in S$. Since there is a path from a_1 to a_2 in $\alpha(\mathcal{D})$ if and only if such an edge

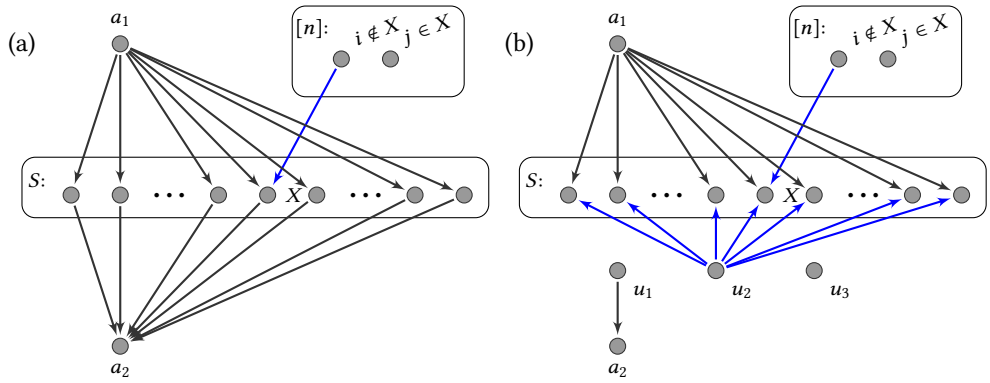

Fig. 7. The graphs from the proof of Theorem 7.3. Edges used for controlling the changes are highlighted in blue.

remains, we conclude that $\bar{a} \in q_{\text{Reach}}(\alpha(\mathcal{D}))$ if and only if $\{i_1, \ldots, i_{k+1}\} \in S$. By Lemma 7.4 therefore $(q_{\text{Reach}}, \{\text{INS}_E, \rho\})$ cannot be maintained in k-ary DynProp, the desired contradiction.

The proof of part (b) is very similar. Here the possible paths from a_1 to a_2 are of the form a_1, X, u_1, a_2. However, initially there are no edges of the form (X, u_1). The first $k+1$ steps of the to-be-constructed change sequence $\alpha(i_1, \ldots i_{k+1})$ yield a situation where at most one node Y is *not* connected to a particular node u_2, and in the final step an edge (Y, u_1) is inserted — but only if $Y \in S$. Altogether, there will again be a path from a_1 to a_2 if and only if $\{i_1, \ldots, i_{k+1}\} \in S$ and Lemma 7.4 yields a contradiction.

Let

- $\rho_1(p_1, p_2) = \mu_1(p_1, p_2; x, y) \stackrel{\text{def}}{=} E(x, y) \vee \big(y = p_1 \wedge E(p_2, x)\big)$ and
- $\rho_2(p_1, p_2, p_3) = \mu_2(p_1, p_2, p_3; x, y) \stackrel{\text{def}}{=} E(x, y) \vee \big(y = p_1 \wedge E(p_2, x) \wedge \neg E(x, p_3)\big)$.

The query ρ_1 inserts edges (x, p_1) for all x with an edge (p_2, x), whereas ρ_2 inserts edges (x, p_1) for all x with an edge (p_2, x) but *without* an edge (x, p_3).

Towards a contradiction, let us assume that $(q_{\text{Reach}}, \{\text{INS}_E, \rho_1, \rho_2\})$ can be maintained in k-ary DynProp, for some k. Again, we aim to use Lemma 7.4 and we let $m = 2$ and $r = 3$ in the statement of the lemma. Let n and $S \subseteq \mathcal{P}_{k+1}([n])$ be arbitrary.

The database $\mathcal{D}$ is a graph with node set $[n] \cup \{a_1, a_2, u_1, u_2, u_3\} \cup S$, where $[n]$, $\{a_1, a_2, u_1, u_2, u_3\}$ and S are pairwise disjoint. The graph has the following edges:

- For each $X \in S$ there are edges (a_1, X) and (u_2, X).
- For each $i \in [n]$ and for each $X \in S$ there is an edge (i, X) if $i \notin X$.
- There is an edge (u_1, a_2).

It can be easily verified that the conditions imposed by Lemma 7.4 are satisfied.

We choose the change sequence $\alpha(x_1, \ldots, x_{k+1})$ as $\rho_1(x_1, u_3), \ldots, \rho_1(x_{k+1}, u_3), \rho_2(u_1, u_2, u_3)$.

For any set $Y = \{i_1, \ldots, i_{k+1}\}$ with $1 \leq i_1 < \cdots < i_{k+1} \leq n$ we define $\alpha_Y \stackrel{\text{def}}{=} \alpha(i_1, \ldots i_{k+1})$. After the first $k+1$ change steps of α_Y, a node X of S is connected to u_3 unless $X = Y$. Therefore, the last change $\rho_2(u_1, u_2, u_3)$ inserts at most one edge, (Y, u_1), but only if $Y \in S$. Thus, in $\alpha_Y(\mathcal{D})$ there is a path from a_1 to a_2 if and only if $Y \in S$.

Application of Lemma 7.4 again yields the desired contradiction.

An inspection of the proofs shows that the graphs used are acyclic and that both constructions immediately work for undirected reachability. □

It remains to give a proof for Lemma 7.4. This proof uses the following combinatorial result which combines Ramsey upper and lower bounds.

Lemma 7.5 ([43, Lemma 2]). *Let $k \in \mathbb{N}$ be arbitrary and τ a k-ary schema. Then there is a monotone and unbounded function $f : \mathbb{N} \mapsto \mathbb{N}$ and an $n_0 \in \mathbb{N}$ such that for every domain A larger than n_0 the following conditions are satisfied:*

(S1) For every τ-database $\mathcal{A}$ over A and every linear order $\prec$ on A there is a subset A' of A of size $|A'| \geq f(|A|)$ such that all $\prec$-ordered k-tuples over A' have the same quantifier-free type in $\mathcal{A}$.

(S2) The set $\mathcal{P}_{k+1}(A)$ can be partitioned into two subsets B and C such that for every set $A' \subseteq A$ of size $|A'| \geq f(|A|)$ there are $b, c \in \mathcal{P}_{k+1}(A')$ with $b \in B$ and $c \in C$.

We remark that the unboundedness of f was not explicitly stated in [43] but it readily follows from its proof in [44], where f is chosen in $\Omega(\log^{(k-1)}(n))$. Monotonicity can be easily achieved.

Furthermore, the proof for Lemma 7.4 makes use of the following *Substructure Lemma*. The lemma exploits that in quantifier-free programs, an auxiliary tuple $\bar{c}$ after an insertion or deletion of a tuple $\bar{d}$ can only depend on those two tuples. We refer to [47] for more intuition. In the following, we denote by $\mathcal{S} \restriction A$ the restriction of a state $\mathcal{S}$ to domain A, that is the state with domain A, in which all (input and auxiliary) relations of $\mathcal{S}$ are restricted to A, in the usual sense.

Let π be an isomorphism from a database $\mathcal{S}$ to a database $\mathcal{T}$. Two changes $\delta = (\rho, \bar{a})$ on $\mathcal{S}$ and $\delta' = (\rho', \bar{b})$ on $\mathcal{T}$ are said to be *π-respecting* if $\rho = \rho'$ and $\bar{b} = \pi(\bar{a})$. Two sequences $\alpha = \delta_1 \cdots \delta_m$ and $\beta = \delta'_1 \cdots \delta'_m$ of changes respect π if δ_i and δ'_i are π-respecting for every $i \leq m$. Recall that $\mathcal{P}_\alpha(\mathcal{S})$ denotes the state obtained by executing the dynamic program $\mathcal{P}$ for the change sequence α from state $\mathcal{S}$.

Lemma 7.6 (Substructure Lemma). *Let $\mathcal{P}$ be a DynProp-program and let $\mathcal{S}$ and $\mathcal{T}$ be states of $\mathcal{P}$ with domains S and T. Further let $A \subseteq S$ and $B \subseteq T$ such that $\mathcal{S} \restriction A$ and $\mathcal{T} \restriction B$ are isomorphic via π. Then $\mathcal{P}_\alpha(\mathcal{S}) \restriction A$ and $\mathcal{P}_\beta(\mathcal{T}) \restriction B$ are isomorphic via π for all π-respecting quantifier-free change sequences α, β on A and B.*

The proof of the Substructure Lemma for single-tuple changes presented in [47] immediately carries over to quantifier-free changes.

Proof (of Lemma 7.4). Let (q, Δ) be a dynamic query where q is an m-ary τ_{in}-query and Δ is a set of quantifier-free replacement queries. Suppose q satisfies the antecedent of the lemma. Towards a contradiction, let us assume that there is a DynProp program $\mathcal{P}$ over a k-ary auxiliary schema τ_{aux} that maintains (q, Δ).

Let τ be the schema $\tau_{\text{in}} \cup \tau_{\text{aux}} \cup \{s_1, \ldots, s_m\} \cup \{t_1, \ldots, t_r\}$, where the s_i and t_i are additional constant symbols. Let $n > n_0$ be arbitrary, where n_0 is as guaranteed by Lemma 7.5 applied to k and τ, and large enough such that $f(n_0) > k + 1$.

Let $A \stackrel{\text{def}}{=} [n] \cup \{a_1, \ldots, a_m, u_1, \ldots, u_r\}$ and let $B, C \subseteq \mathcal{P}_{k+1}(A)$ be the partition guaranteed to exist by property (S2) of Lemma 7.5. We choose S as B.

Let $\mathcal{D}$ and α be as in the condition of the statement of the lemma. Let $\mathcal{A}_\mathcal{D}$ be the auxiliary database over schema $\tau_{\text{aux}} \cup \{s_1, \ldots, s_m\} \cup \{t_1, \ldots, t_r\}$ that is obtained by $\mathcal{P}$ when the tuples of $\mathcal{D}$ are inserted in some arbitrary order to an initially empty database, and in which the constant symbols $s_1, \ldots, s_m, t_1, \ldots, t_r$ are interpreted by $a_1, \ldots, a_m, u_1, \ldots, u_r$, respectively. Furthermore, let $\mathcal{A}$ be the database induced by A from $(\mathcal{D}, \mathcal{A}_\mathcal{D})$. Let A be ordered in the natural way such that all elements from $[n]$ precede all other elements. Let A' be the subset of A as guaranteed by property (S1) of Lemma 7.5. Since all ordered tuples of A' have the same type, they can not involve constants and thus $A' \subseteq [n]$. Now property (S2) of Lemma 7.5 guarantees that there exist sets $b, c \in \mathcal{P}_{k+1}(A')$

with $b \in B$ and $c \in C$. Let $i_1 < \cdots < i_{k+1}$ and $j_1 < \cdots < j_{k+1}$ be such that $b = \{i_1, \ldots, i_{k+1}\}$ and $c = \{j_1, \ldots, j_{k+1}\}$.

We note that both sequences $\bar{i} = i_1, \ldots, i_{k+1}$ and $\bar{j} = j_1, \ldots, j_{k+1}$ fulfil the requirements stated in the lemma and furthermore, $(\mathcal{D}, \bar{a}, \bar{i}, \bar{u}) \equiv_0 (\mathcal{D}, \bar{a}, \bar{j}, \bar{u})$. Therefore Lemma 7.6 guarantees that $(\alpha(\bar{i}, \bar{u})(\mathcal{D}), \bar{a}, \bar{u}) \equiv_0 (\alpha(\bar{j}, \bar{u})(\mathcal{D}), \bar{a}, \bar{u})$ holds as well. We can conclude that $\bar{a} \in q(\alpha(\bar{i}, \bar{u})(\mathcal{D}))$ if and only if $\bar{a} \in q(\alpha(\bar{j}, \bar{u})(\mathcal{D}))$ and finally, by the property assumed in the statement of the lemma, that $b \in S$ if and only if $c \in S$. However $c \notin S$, since $S = B$, the desired contradiction. □

8 PARAMETER-FREE CHANGES

In this section we study replacement queries without parameters on ordered databases, or equivalently, changes expressible as relational algebra queries that do not use constant values. It turns out that in this setting a large class of queries can be maintained in DynFO: all queries that can be expressed in uniform AC^1 and thus, in particular, all queries that can be answered in logarithmic space. This result exploits the fact that for a fixed set of replacement queries without parameters there is only a constant number of possible changes to a database, in each step.

One might suspect that parameter-free replacement queries are not very powerful, especially when they are applied to the initially empty input database. However, when the input database comes with a built-in linear order, one can actually construct every finite graph with relatively simple replacement queries (and similarly for other kinds of databases). For instance, one can cycle through all pairs of nodes in lexicographic order. If (u, v) is the current maximal pair, operation *keep* can move to $(u, v + 1)$ (inserting it into E) while leaving (u, v) in E and *drop* can move to $(u, v + 1)$ while taking (u, v) out from E. For this reason, including a linear order into the input database will yield stronger results. This motivates the following definitons.

An *ordered* database $\mathcal{D}$ contains a built-in linear order $\leq$ on its domain that is not modified by any change. We identify elements of an ordered database with numbers. In the following, the minimal element with respect to $\leq$ is considered as 0.

The update programs constructed in this section use, as additional auxiliary relation, a binary BIT-relation containing all pairs (i, j) of numbers, for which the i-th bit of the binary representation of j is 1. From a linear order, the BIT relation can be easily computed by an initialization procedure. In practical scenarios one typically has access to the binary encoding of the input and it is not necessary to introduce an additional relation.

By AC^1 we denote the class of queries that can be computed by a uniform[24] family of circuits of "and", "or" and "not" gates with polynomial size, depth $O(\log n)$ and unbounded fan-in. It is well known that AC^1 contains all queries from LOGSPACE and NL. We can now state the main theorem of this section.

Theorem 8.1. *Let q be an AC^1 query over ordered databases and Δ a finite set of parameter-free first-order definable replacement queries. Then (q, Δ) is in DynFO with suitable initialization.*

Proof. We first explain the idea underlying the proof.

It uses the characterization of AC^1 by iterated first-order formulas. More precisely, we use the equality $AC^1 = \text{IND}[\log n]$ from [26, Theorem 5.22]. The class $\text{IND}[t(n)]$ contains a query q if it can be defined by a first-order formula ψ_q that may use a relation R obtained by $O(t(n))$-fold iteration of a first-order formula φ_R, where n is the size of the domain. We only give an example and refer to [26, Definition 4.16] for a formal definition.[25] Consider the formula $\varphi_R(x, y) = (x = y) \vee E(x, y) \vee \exists z\big(R(x, z) \wedge R(z, y)\big)$. When applying the formula to a graph and an empty relation R

[24] for concreteness: first-order uniform [26]

[25] In the setting of [26], first-order formulas may use built-in relations $\leq$ and BIT. The relation $\leq$ is also present here, the relation BIT can be generated by a suitable initialization, see [26, Exercise 4.18]. Furthermore Immerman's definition of

it defines the relation R_1 of paths of length 1, applying it to $R \stackrel{\text{def}}{=} R_1$ defines the paths of length 2; in general applying the formula to $R \stackrel{\text{def}}{=} R_i$ defines the paths of length 2^i. Thus $\log n$-fold application of φ_{R} defines the transitive closure relation of a graph with n vertices and therefore q_{Reach} is in IND[$\log n$] (by choosing $\psi_{q_{\mathrm{Reach}}} = R$). For a database $\mathcal{D}$, we write $\varphi^i(\mathcal{D})$ for the relation resulting from i-fold application of φ to $(\mathcal{D}, \emptyset)$.

Let q be a query in AC^1 and let k be such that q is first-order definable by ψ_q from a relation R that is obtained by $k \log n$ applications of a formula φ_R.

The program $\mathcal{P}$ that we construct maintains R using a technique inspired from prefetching, which was called *squirrel technique* in [46]. The result of q is then extracted from R by ψ_q. For maintaining R, the program starts a thread θ_β at any point t in time and for each possible future sequence β of $2 \log n$ change operations. Here and in the following, we count the occurrence of one change as one time step.

Within the next $\log n$ steps (i.e. changes), it always compares whether the actual change sequence α is the prefix of β of length $\log n$. If not, thread θ_β is abandoned, as soon as α departs from β. For each of these $\log n$ steps, θ_β simulates two change operations of β and applies them to the database $\mathcal{D}_t$ at time t, consecutively. After $\log n$ steps, that is, at time $t + \log n$, thread θ_β has computed the *target database* $\beta(\mathcal{D}_t)$.

During the next $\log n$ steps until time $t + 2 \log n$, θ_β evaluates q on $\beta(\mathcal{D}_t)$ by repeatedly applying the formula φ_R, k times during each single step. Again, if the actual change sequence departs from β then θ_β is abandoned. However, if β is the actual change sequence from time t to $t + 2 \log n$, the thread θ_β can extract the correct query result $q(\beta(\mathcal{D}_t))$ from R at time $t + 2 \log n$.

We note that, although the time window in the above sketch stretches over $2 \log n$ change operations from time t to $t + 2 \log n$, the actual sequences whose effect on the current database is precomputed are never longer than $\log n$. This is because the application of all $2 \log n$ operations of a sequence takes until time $t + \log n$ and by that time the first $\log n$ of these operations already lie in the past. In fact, the program $\mathcal{P}$ only needs to maintain $\beta(\mathcal{D}_t)$ for all sequences β of length exactly $\log n$, as soon as $t \geq \log n$.

Of course, $\mathcal{P}$ uses many threads at any time. However, this is feasible, because only a constant number, $d = |\Delta|$, of change operations is available at any time (and there are no parameters). More precisely, there are only $d^{2 \log n} = 2^{2 \log d \log n} = n^{2 \log d}$ many different change sequences of length $2 \log n$, each of which can be encoded by a tuple of arity $2 \log d$ over the domain.

So far, our sketch explains how $\mathcal{P}$ can give correct answers for all times $t \geq 2 \log n$. All previous time points have to be dealt with by the initialization. This initialization also equips the program with the BIT relation. Clearly, the initialization can be computed in AC^1, and therefore also in polynomial time.

We next present a more detailed proof, but for simplicity, we only consider the case of ordered graphs ($\tau_{\mathrm{in}} = \{E, \leq\}$), $d = 2$ and $k = 1$. That is, there are only two change operations, ρ_0 and ρ_1, and R can be obtained by $\log n$ applications of φ_R. The proof can easily be generalized to the case of databases over other schemas and for arbitrary d and k.

Let G be an ordered graph with n vertices. For simplicity we assume that that the vertex set of G is just $\{0, \ldots, n-1\}$, that $\leq$ is the natural linear order, and that n is a power of 2. The latter assumption guarantees that $\log n$ is a natural number and that bit sequences of length $\log n$ can be represented by one node of G using BIT. For the general case, bit sequences of length $\lceil \log n \rceil$ can be represented by two nodes.

IND[$t(n)$] allows for nested iterations. However, the proof of [26, Lemma 5.3] yields a normal form of IND[$t(n)$] with only one iterated formula.

We encode change sequences by elements of the domain as follows: A sequence $\beta = \delta_1 \cdots \delta_{\log n}$ is encoded by the node w_β, whose bit string representation has 1 at position i if and only if $\delta_i = \rho_1$. We denote the change sequence encoded by node w as β_w.

We first describe the auxiliary relations needed for time points $t \geq 2 \log n$. In the following, we assume that relations for the arithmetic operations $+, \times$ and the BIT-predicate are provided by the initialization.[26]

For a time t, we denote by G_t the graph at time t. Likewise, we denote the value of any (input or auxiliary) relation S at time t by S_t.

- Relation F consists of tuples of the form (i, w, u, v) and encodes, for each $i \leq \log n$ and each change sequence β of length $\log n$ the graph $\beta(G_{t-i})$, as follows. We denote the edge relation induced by F for any i and w by $F^{i,w} = \{(u, v) \mid (i, w, u, v) \in F\}$. Then, for each $i \leq \log n$ and each change sequence β of length $\log n$, F_t^{j,w_β} shall be the edge relation of $\beta(G_{t-i})$.
- Relation T consists of tuples of the form $(i, w, \bar{a})$ and represents the temporary result of R after some applications of φ_R to a modified graph. We denote the intermediate relation represented by T for some i and w by $T^{i,w} = \{\bar{a} \mid (i, w, \bar{a}) \in T\}$. Then, for every $i \leq \log n$ and each change sequence β of length $\log n$, it shall hold $T_t^{i,w_\beta} = \varphi_R^i(\beta(G_{t-i}))$.
- The history relation H always contains one element w which encodes the sequence β_w of changes that occurred during the most recent $\log n$ many change steps.
- The query relation Q.

The query relation Q_t can be inferred from the relations T_t and H_t using the formula ψ_q, since $R_t = \varphi_R^{\log n}(\beta_w(G_{t-\log n})) = T^{\log n, w}$ where w is the unique $w \in H_t$.

It is easy to maintain H in a first-order fashion, and it thus only remains to describe how F and T can be maintained.

For every i and w, $F_{t+1}^{i,w}$ can be easily obtained from F_t and H_t as follows. Let $\beta_w = \delta_1 \cdots \delta_{\log n}$ and let δ_0 be the actual change step that occurred at time $t - i$, as encoded in the single tuple of H_t. Then, $F_{t+1}^{i,w} = \beta_w(G_{t+1-i}) = \beta_w(\delta_0(G_{t-i})) = \delta_{\log n}(\beta_{w'}(G_{t-i})) = \delta_{\log n}(F_t^{i,w'})$, where w' encodes $\beta' = \delta_0 \cdots \delta_{\log n - 1}$. For every i and w, $T_{t+1}^{i,w} = \varphi_R^i(\beta(G_{t+1-i})) = \varphi_R(\varphi_R^{i-1}(\beta(G_{t-(i-1)}))) = \varphi_R(T_t^{i-1,w})$. In both cases, the first-order maintainability is immediate.

□

As already discussed in Section 3, the previous result translates to a variant of the FOIES framework of [16]. Here, the change operations `add` and `remove` may not have the inserted or deleted element as parameter, but instead add some new element to the domain as the largest element of $\leq$ or remove that element, respectively. The proof can be easily be adapted for this setting.

9 CONCLUSION

In this article we continued the study of incremental view maintenance from a declarative point of view. Even though in applications incremental changes to databases often involve sets of tuples, previous work in the area has been focussed on single-tuple changes. As a step towards a better understanding of complex changes we provided a formalization of declaratively defined change operations in terms of first-order defined replacement queries, and studied which results can be transferred to this more general framework.

The main insight of our study is that many maintainability results carry over from the single-tuple world to more general change operations. We were actually quite surprised to see that so many

[26] In fact, the BIT-predicate is sufficient, as $+$ and $\times$ are FO-definable from BIT and the linear order [26, Theorem 1.17].

positive results survive this transition. Our focus was on the undirected reachability query. This query is in DynFO even under first-order definable insertions. For insertions defined by well-studied classes of queries such as conjunctive queries, the dynamic programs are of practically feasible size. We reported on a prototypical implementation of these programs in SQL. In almost all test scenarios, the implemented dynamic program for complex changes achieved better runtimes, compared to a dynamic program processing complex changes as sequences of single-edge changes, and evaluation from scratch with built-in mechanisms of database systems. Encouraged by those results we plan to evaluate the dynamic program for directed reachability presented in [4] empirically.

Many questions regarding maintainability under complex changes remain open, for instance: To which extent can the reachability query for (undirected or acyclic) graphs be maintained under definable deletions? What about reachability for unrestricted directed graphs under definable insertions? What about other queries?

We were less surprised by the fact that stronger change operations can yield inexpressibility, but even these results required some care. Our main contribution in that respect is the proof that DynProp cannot maintain the reachability query under very simple quantifier-free replacement queries.

From Theorem 8.1 about parameter-free changes and its proof, we take another insight: the squirrel technique is quite powerful to prepare an update program for a non-constant (i.e., logarithmic) number of changes. In [6] a similar technique was exploited to prove that queries definable in monadic second-order logic can be maintained on graphs of bounded treewidth. Another consequence is that inexpressibility proofs need to take the squirrel technique into account and to argue "around it".

ACKNOWLEDGMENTS

We thank Dennis Ciba for his work on the implementation and the framework for the empirical evaluation, and the Database and Information Systems group at TU Dortmund led by Jens Teubner for providing access to a compute server for the experiments. We also thank the anonymous referees for very valuable suggestions for improving a draft of this article.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley, Boston, MA, USA. http://webdam.inria.fr/Alice/

[2] Tom J. Ameloot, Jan Van den Bussche, and Emmanuel Waller. 2013. On the Expressive Power of Update Primitives. In *Proceedings of the 32Nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '13).* ACM, New York, NY, USA, 139–150. https://doi.org/10.1145/2463664.2465218

[3] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. 2017. gMark: Schema-Driven Generation of Graphs and Queries. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 856–869. https://doi.org/10.1109/TKDE.2016.2633993

[4] Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. 2015. Reachability is in DynFO. In *Automata, Languages, and Programming - 42nd International Colloquium (ICALP), Proceedings, Part II.* Springer-Verlag, Berlin, Heidelberg, 159–170. https://doi.org/10.1007/978-3-662-47666-6_13

[5] Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. 2015. Reachability is in DynFO. *CoRR* abs/1502.07467 (2015). arXiv:arXiv:abs/1502.07467 http://arxiv.org/abs/1502.07467

[6] Samir Datta, Anish Mukherjee, Thomas Schwentick, Nils Vortmeier, and Thomas Zeume. 2017. A Strategy for Dynamic Programs: Start over and Muddle Through. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland (LIPIcs)*, Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl (Eds.), Vol. 80. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 98:1–98:14. https://doi.org/10.4230/LIPIcs.ICALP.2017.98

[7] Samir Datta, Anish Mukherjee, Nils Vortmeier, and Thomas Zeume. 2018. Reachability and Distances under Multiple Changes. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic (LIPIcs)*, Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella

(Eds.), Vol. 107. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 120:1–120:14. https://doi.org/10.4230/LIPIcs.ICALP.2018.120

[8] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F Italiano. 2010. Dynamic graph algorithms. In *Algorithms and theory of computation handbook*. Chapman & Hall/CRC, 9–9.

[9] Guozhu Dong, Leonid Libkin, Jianwen Su, and Limsoon Wong. 1999. Maintaining Transitive Closure of Graphs in SQL. *Int. Journal of Information Technology* 51, 1 (1999), 46–78.

[10] Guozhu Dong, Leonid Libkin, and Limsoon Wong. 2003. Incremental recomputation in local languages. *Inf. Comput.* 181, 2 (2003), 88–98. https://doi.org/10.1016/S0890-5401(03)00017-8

[11] Guozhu Dong and Chaoyi Pang. 1997. Maintaining Transitive Closure in First Order After Node-Set and Edge-Set Deletions. *Inf. Process. Lett.* 62, 4 (1997), 193–199. https://doi.org/10.1016/S0020-0190(97)00066-5

[12] Guozhu Dong and Jianwen Su. 1993. First-Order Incremental Evaluation of Datalog Queries. In *Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages (DBPL-4)*. 295–308.

[13] Guozhu Dong and Jianwen Su. 1995. Incremental and Decremental Evaluation of Transitive Closure by First-Order Queries. *Inf. Comput.* 120, 1 (1995), 101–106. https://doi.org/10.1006/inco.1995.1102

[14] Guozhu Dong and Jianwen Su. 1997. Deterministic FOIES are Strictly Weaker. *Ann. Math. Artif. Intell.* 19, 1-2 (1997), 127–146. https://doi.org/10.1023/A:1018951521198

[15] Guozhu Dong and Jianwen Su. 1998. Arity Bounds in First-Order Incremental Evaluation and Definition of Polynomial Time Database Queries. *J. Comput. Syst. Sci.* 57, 3 (1998), 289–308. https://doi.org/10.1006/jcss.1998.1565

[16] Guozhu Dong, Jianwen Su, and Rodney W. Topor. 1995. Nonrecursive Incremental Evaluation of Datalog Queries. *Ann. Math. Artif. Intell.* 14, 2-4 (1995), 187–223. https://doi.org/10.1007/BF01530820

[17] Guozhu Dong and Rodney W. Topor. 1992. Incremental Evaluation of Datalog Queries. In *Proceedings of the 4th International Conference on Database Theory (ICDT)*. 282–296. https://doi.org/10.1007/3-540-56039-4_48

[18] Kousha Etessami. 1998. Dynamic Tree Isomorphism via First-Order Updates. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 235–243. https://doi.org/10.1145/275487.275514

[19] Solomon Feferman. 1957. Some recent work of Ehrenfeucht and Fraïssé. In *Proc. Summer Institute of Symbolic Logic*. 201–209.

[20] Solomon Feferman and Robert L. Vaught. 1959. The first order properties of algebraic systems. *Fund. Math.* 47 (1959), 57–103.

[21] Wouter Gelade, Marcel Marquardt, and Thomas Schwentick. 2012. The dynamic complexity of formal languages. *ACM Trans. Comput. Log.* 13, 3 (2012), 19. https://doi.org/10.1145/2287718.2287719

[22] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2001. The complexity of acyclic conjunctive queries. *J. ACM* 48, 3 (2001), 431–498.

[23] Erich Grädel and Sebastian Siebertz. 2012. Dynamic definability. In *15th International Conference on Database Theory (ICDT)*. 236–248. https://doi.org/10.1145/2274576.2274601

[24] Ashish Gupta and Inderpal Singh Mumick. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18. http://sites.computer.org/debull/95JUN-CD.pdf

[25] William Hesse and Neil Immerman. 2002. Complete Problems for Dynamic Complexity Classes. In *17th IEEE Symposium on Logic in Computer Science (LICS), Proceedings*. 313. https://doi.org/10.1109/LICS.2002.1029839

[26] Neil Immerman. 1999. *Descriptive complexity*. Springer. https://doi.org/10.1007/978-1-4612-0539-5

[27] Christoph Koch. 2010. Incremental query evaluation in a ring of databases. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 87–98. https://doi.org/10.1145/1807085.1807100

[28] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.* 23, 2 (2014), 253–278. https://doi.org/10.1007/s00778-013-0348-4

[29] Leonid Libkin. 2004. *Elements of Finite Model Theory*. Springer.

[30] Ling Liu and M Tamer Özsu. 2018. *Encyclopedia of database systems*. Springer.

[31] Johann A. Makowsky. 2004. Algorithmic uses of the Feferman-Vaught Theorem. *Ann. Pure Appl. Logic* 126, 1-3 (2004), 159–213. https://doi.org/10.1016/j.apal.2003.11.002

[32] Chaoyi Pang, Guozhu Dong, and Kotagiri Ramamohanarao. 2005. Incremental maintenance of shortest distance and transitive closure in first-order logic and SQL. *ACM Trans. Database Syst.* 30, 3 (2005), 698–721. https://doi.org/10.1145/1093382.1093384

[33] Sushant Patnaik and Neil Immerman. 1994. Dyn-FO: A Parallel, Dynamic Complexity Class. In *PODS 1994*. 210–221.

[34] Sushant Patnaik and Neil Immerman. 1997. Dyn-FO: A Parallel, Dynamic Complexity Class. *J. Comput. Syst. Sci.* 55, 2 (1997), 199–209. https://doi.org/10.1006/jcss.1997.1520

[35] Piotr Przymus, Aleksandra Boniewicz, Marta Burzańska, and Krzysztof Stencel. 2010. Recursive query facilities in relational databases: a survey. In *Database Theory and Application, Bio-Science and Bio-Technology*. Springer, 89–99.

[36] Thomas Schwentick, Nils Vortmeier, and Thomas Zeume. 2017. Dynamic Complexity under Definable Changes. In *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy (LIPIcs)*, Michael Benedikt and Giorgio Orsi (Eds.), Vol. 68. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 19:1–19:18. https://doi.org/10.4230/LIPIcs.ICDT.2017.19

[37] Thomas Schwentick, Nils Vortmeier, and Thomas Zeume. 2017. Dynamic Complexity under Definable Changes. *CoRR* abs/1701.02494 (2017). http://arxiv.org/abs/1701.02494

[38] Thomas Schwentick and Thomas Zeume. 2016. Dynamic complexity: recent updates. *SIGLOG News* 3, 2 (2016), 30–52. https://doi.org/10.1145/2948896.2948899

[39] Galina Shalygina and Boris Novikov. 2017. Implementing Common Table Expressions for MariaDB. In *Second Conference on Software Engineering and Information Management (SEIM-2017)*. 12–17.

[40] Sebastian Siebertz. 2011. *Dynamic Definability*. Diploma Thesis. RWTH Aachen.

[41] Dimitra Vista. 1998. Integration of Incremental View Maintenance into Query Optimizers. In *Advances in Database Technology - EDBT'98, 6th International Conference on Extending Database Technology, Valencia, Spain, March 23-27, 1998, Proceedings (Lecture Notes in Computer Science)*, Hans-Jörg Schek, Fèlix Saltor, Isidro Ramos, and Gustavo Alonso (Eds.), Vol. 1377. Springer, 374–388. https://doi.org/10.1007/BFb0100997

[42] Volker Weber and Thomas Schwentick. 2007. Dynamic Complexity Theory Revisited. *Theory Comput. Syst.* 40, 4 (2007), 355–377. https://doi.org/10.1007/s00224-006-1312-0

[43] Thomas Zeume. 2014. The Dynamic Descriptive Complexity of k-Clique. In *Mathematical Foundations of Computer Science (MFCS) - 39th International Symposium, Proceedings, Part I*. 547–558. https://doi.org/10.1007/978-3-662-44522-8_46

[44] Thomas Zeume. 2015. *Small Dynamic Complexity Classes*. Ph.D. Dissertation. TU Dortmund University.

[45] Thomas Zeume. 2017. The dynamic descriptive complexity of k-clique. *Inf. Comput.* 256 (2017), 9–22. https://doi.org/10.1016/j.ic.2017.04.005

[46] Thomas Zeume and Thomas Schwentick. 2014. Dynamic Conjunctive Queries. In *Proc. 17th International Conference on Database Theory (ICDT)*. 38–49. https://doi.org/10.5441/002/icdt.2014.08

[47] Thomas Zeume and Thomas Schwentick. 2015. On the quantifier-free dynamic complexity of Reachability. *Inf. Comput.* 240 (2015), 108–129. https://doi.org/10.1016/j.ic.2014.09.011