Sketches of Dynamic Complexity

Thomas Schwentick
TU Dortmund
thomas.schwentick@tu-dortmund.de

Nils Vortmeier
TU Dortmund
nils.vortmeier@tu-dortmund.de

Thomas Zeume
Ruhr University Bochum
thomas.zeume@rub.de

ABSTRACT

How can the result of a query be updated after changing a database? This is a fundamental task for database management systems which ideally takes previously computed information into account. In dynamic complexity theory, it is studied from a theoretical perspective where updates are specified by rules written in first-order logic.

In this article we sketch recent techniques and results from dynamic complexity theory with a focus on the reachability query.

1. INTRODUCTION

Assume you are running a very traditional relational DBMS that supports all queries that can be expressed in the relational algebra, but nothing else.[1] Then you precisely understand what kinds of queries you can pose and which you cannot: you are limited to queries that are expressible in first-order logic.

You think that it might be helpful that you are interested in continuously asking the same query. Maybe the database you maintain is actually a graph database and you would be interested to evaluate a fixed set of regular path queries all over again. Maybe you also know that changes to your database are not very frequent. Is there a way to cope with your queries without writing programs or installing that graph database engine?

This is the setting that is assumed in this article and the setting of dynamic complexity as introduced by Patnaik and Immerman [34] and similarly by Dong and Su [13] in the early nineties: there is an initially empty database, tuples can be inserted and deleted and after each change of the database, the answer to some fixed query needs to be computed with first-order logic means. Besides the "real" relations, the database can have additional, auxiliary relations, one of which always represents the query answer to the standing query. After each change step your database can apply first-order queries to update these auxiliary relations.

This setting is similar to other typical database settings, but it differs from a typical incremental query maintenance setting in that it addresses queries that are *not* expressible in the relational algebra, and from a typical view maintenance setting because auxiliary relations are allowed[2].

In this article, we want to report on some progress that dynamic complexity has seen during the last years. Besides the result that reachability on directed graphs *can* be maintained in this framework, much of the research has focussed on new techniques and the extension of the framework towards bulk changes, as opposed to single-tuple changes. The ability to maintain regular path queries is one outcome of this line of work.

We develop the framework incrementally while giving sketches[3] of some recent and some older key results and techniques. Most of these results concern the reachability query REACH on directed or undirected graphs. This query maps a graph $G = (V, E)$ to the transitive closure of the edge relation E. In other words, REACH(G) is the binary relation that contains a pair (u, v) of nodes if there is a non-empty path from u to v in G. An immediate consequence of these results for maintaining reachability is that regular path queries can be maintained as well, see Sketch 10.

In this article we borrow from several talks we presented in the last few years as well as from some of our articles [8, 11, 10, 9]. For recent, more complete expositions of the current state of the art of dynamic complexity we refer to [40, 43].

2. MAINTAINING REACHABILITY

We start with the very simple scenario where only single edges can be inserted into the graph (database).

Sketch 1:

Single-edge insertions into directed graphs

Since we aim at updating the standing query REACH, the transitive closure of the edge relation is stored as

[1]This assumption is not entirely realistic but very common in foundational database research.

[2]We emphasise that recent higher-order incremental view maintenance frameworks also use auxiliary views [29, 33].

[3]As in "brief description", not as used in, e.g., streaming algorithms.

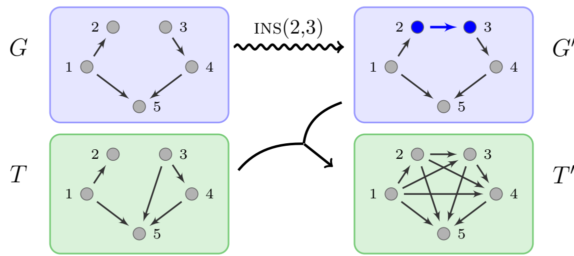

Figure 1: The dynamic scenario. After inserting edge $(2, 3)$ there is a path from $x = 1$ to $y = 4$ thanks to the previously existing paths from 1 to 2 and from 3 to 4.

an auxiliary relation.

How can we update the transitive closure of a graph after inserting a single edge? After inserting an edge (u, v), there is a path from a node x to a node y if there has been a path before the insertion or if there were paths from x to u and from v to y, cf. Figure 1. Thus, if T denotes the auxiliary relation that stores the transitive closure of E, the update that needs to be applied can be specified as follows.

on insert (u, v) **update** T **as**

$$T'(x, y) \stackrel{\text{def}}{=} T(x, y) \vee \big(T(x, u) \wedge T(v, y)\big)$$

The semantics is that, after inserting the edge (u, v), the relation T is replaced according to the query $T(x, y) \vee \big(T(x, u) \wedge T(v, y)\big)$.

We call the above rule an *update rule*. A *dynamic (first-order) program* can use finitely many auxiliary relations $R_1, \ldots, R_m$ and provides a *(first-order) update rule* for each of these relations and each admissible change operation. In the above case, the only admissible *change operation* is insertion of edges.[4] Each update rule can access the edge relation and (the current versions of the) relations $R_1, \ldots, R_m$.

Most often, admissible change operations are insertions or deletions of edges. But the exact form, e.g., whether single-tuple or bulk changes are allowed and how they are specified, depends on the context. When an actual change occurs, the program updates its auxiliary relations by simultaneously applying their respective update rules for the underlying change operation.

A dynamic program *maintains* the result of a query if some designated auxiliary relation stores the result of the query after all possible sequences of admissible changes. As an example, the above (single-rule) program maintains the query Reach on directed graphs under single-edge insertions. We emphasise that this particular rule does not even use quantifiers.

[4]We note that each *change operation* can be instantiated by actual *changes*, e.g., the insertion of a concrete edge.

The class DynFO consists of all pairs $(\mathcal{Q}, \Delta)$ such that the query $\mathcal{Q}$ can be maintained by a dynamic first-order program under the set Δ of admissible changes. For a pair $(\mathcal{Q}, \Delta) \in$ DynFO we usually say that $\mathcal{Q}$ is in DynFO under Δ-change operations. As we have just seen, Reach is in DynFO under single-edge insertions.

2.1 Undirected Reachability: from single to bulk changes

Allowing insertions *and* deletions offers "full change power" in the sense that each graph can be transformed into each other graph (with the same vertex set). The question whether Reach can be maintained when edges can be inserted *and* deleted had been a driving force for research in dynamic complexity for twenty years. It turned out that reachability under edge insertions and deletions cannot be maintained in the same simple fashion as in the insertion-only case, and we present some early-known barriers in Section 3.

For now, we concentrate on the easier case of maintaining reachability for undirected graphs. We show how this query can be maintained under insertions and deletions of single edges, and generalise this result to more complex changes. We will come back to reachability for directed graphs in Subsection 2.2.

Besides its elementary update rule, reachability under edge insertions is simple in another sense: it only needs the query answer relation itself as auxiliary relation, i.e., the transitive closure of the edge relation. Trying to maintain a query without any further relations than the query relation itself is a natural first step in the search for a dynamic program. Unfortunately this does not work out for undirected reachability under insertions and deletions [14, Theorem 5.7]. Intuitively, this is because the transitive closure might not yield much information. For instance, the transitive closure of a cycle is a full binary relation which is not helpful for deriving the new transitive closure after deleting two edges from the cycle.

If, as in this case, the query relation does not suffice, one often sees what kind of information is "missing" and one can try to maintain the query by adding another auxiliary relation. This approach could be termed *iterated wishing*: to maintain a certain query, you might wish you had a certain auxiliary relation R_1 available, so you assume you have it, and then you check whether also R_1 can be maintained; if you fail, you wish for more helpful auxiliary relations, and so on.

Sketch 2:

Single-edge insertions and deletions in undirected graphs

After seeing that the transitive closure itself does not suffice, it seems natural to "wish for" a spanning for-

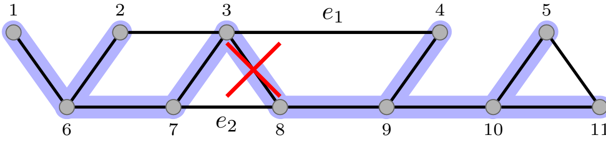

Figure 2: Deleting a single edge from an undirected graph. A spanning forest is highlighted in blue. After removal of $(3, 8)$, edges $(3, 4)$ and $(7, 8)$ are potential new edges for the spanning forest, of which the lexicographically smallest edge $(3, 4)$ is chosen by the update rule.

est in order to maintain reachability for undirected graphs under insertions *and deletions*. It turns out that one more wish is helpful: its accompanying in-between relation.[5] More precisely, the following two auxiliary relations can be used to maintain REACH on undirected graphs [35, Theorem 4.1]:

- a binary relation F with $(a, b) \in F$ if (a, b) is an edge of a (fixed) spanning forest; and
- a ternary relation B with $(a, b, c) \in B$ if b is in between a and c in the spanning forest stored in F, so, if a and c are in the same connected component and b is part of the unique path from a to c in the spanning forest.

We now have to verify that F and B can be updated after inserting or deleting an edge, thereby establishing the following result [35, Theorem 4.1].

PROPOSITION 2.1. REACH *on undirected graphs is in* DYNFO *under single-edge insertions and deletions.*

Updating the auxiliary relations after edge insertions is very similar to the approach of Sketch 1: if the new edge connects two connected components then it is added to the spanning forest F, otherwise nothing changes. Likewise, deletions of edges not in F are easy to handle, as they leave the auxiliary relations unchanged. We therefore focus on deletions of edges of the spanning forest, see Figure 2.

The formula

$$\psi(x, y, u, v) = E(x, y) \wedge B(x, u, y) \wedge B(x, v, y)$$

expresses that the nodes x and y of the edge (x, y) are in different connected components of the spanning forest after removing (u, v) and thus (x, y) is a candidate for "repairing" the spanning forest.

However, only one such edge can be added to the spanning forest and therefore some tie-breaker is needed. To this end, it is helpful to maintain a linear order on the vertices and to add the lexicographically smallest edge. The linear order is yet another auxiliary relation one can wish for and which can be updated easily: since the edge relation is initially empty, a linear order on the non-isolated nodes can be built based on the order of edge insertions [17]. In fact, not only a linear order can be established in this way, but also 3-ary relations that encode the corresponding addition and multiplication operations.[6]

So far we only considered *simple change operations*, that is, single-tuple changes. This is a typical model, not only in dynamic complexity, but also in dynamic algorithms. However, to deal with realistic scenarios in particular in database contexts, it would be helpful to maintain queries under more complex change operations. That is, change operations should be able to insert or delete sets of tuples.

Obviously, one can not hope for "arbitrary changes": if they were allowed, then one could produce any arbitrary graph in one step from the empty graph.[7] Thus a query can only be maintainable under arbitrary changes if it can actually be explicitly expressed, statically.

Therefore, one has to lower expectations and restrict complex change operations in one way or another. We will consider size-restricted change operations later on, but start here with *first-order definable* change operations. An *insertion query* is specified by a first-order formula $\varphi(x, y, \bar{z})$ and a tuple $\bar{c}$ of elements. It defines the set of edges that are inserted into the edge relation by the set of all tuples (a, b) that satisfy the formula $\varphi(a, b, \bar{c})$. We emphasise that there is no a priori bound on the number of edges that are inserted in such a step.

From a databases point of view, first-order definable change operations (in the spirit of SQL updates) are a very natural kind of complex change operations.

Sketch 3:

Definable insertions into undirected graphs

It turns out that the reachability query on undirected graphs can be maintained under single-edge deletions and first-order definable insertions. More precisely, the result is as follows [39, Theorem 4.2].

THEOREM 2.2. *For each finite set* Δ *of insertion queries,* REACH *on undirected graphs is in* DYNFO *under single-edge deletions and under insertions defined by the queries in* Δ.

[5] In fact, it can be seen from the proof of [14, Theorem 5.7] that a spanning forest alone is also not sufficient.

[6] That is, e.g., if a is the smallest node and b the second smallest node with respect to this order, then the triple (a, a, b) is in the ternary relation for the corresponding addition, basically encoding $1 + 1 = 2$.

[7] We recall that the empty graph has nodes but not edges.

The dynamic program uses the spanning forest approach as presented in Sketch 2 and relies on a very simple observation, illustrated by Figure 3: if there is a new path between nodes u and v after a first-order defined insertion, then there is such a path in which the number of new edges is bounded by a constant m that only depends on the quantifier depth of the formula defining the insertion. Therefore, for checking whether there is a path between u and v, a first-order update rule can guess at most m newly inserted edges and combine them with previously existing paths.

In general, the constant m can be large, but if the insertion formula is a union of ℓ conjunction queries, it is bounded by 2ℓ [39, Proposition 4.3]. An evaluation of a prototypical implementation [39, Section 5] shows that dynamic programs for insertions defined by small unions of conjunctive queries perform well in some scenarios in comparison with other methods of answering Reach on undirected graphs.

Another obvious restriction of complex changes is to bound the number of edges that can be inserted or deleted in one step. We next consider the insertion of sets of edges as operation and restrict it to sets of $\mathcal{O}(\log n)$ many edges. We assume that a linear order and its corresponding addition and multiplication relations are given as "built-in relations", since they cannot be computed incrementally as before. We make this assumption transparent and write DynFO$(+, \times)$ for the class of queries that are maintained by dynamic programs with access to built-in $\leq$, $+$ and $\times$.

The technique used for such operations can be understood as a *simulation of monadic second-order logic*. Monadic second-order logic MSO extends first-order logic by quantification over sets. The basic idea of the simulation is that a subset of a set of logarithmically many nodes can be encoded by one node, since a node basically corresponds to a bit string of length $\log n$. Therefore, set quantification over such small sets can be simulated by node quantification over the full graph. The connection between node subsets and nodes can be drawn with the help of the built-in linear order and the arithmetic relations $+$ and $\times$, as the bit string representation of a node can be expressed from them by first-order formulas, see [26, Theorem 1.17].

Sketch 4:

Log-size insertions to undirected graphs

It turns out that reachability on undirected graphs can be maintained under single-edge deletions and insertions of $\mathcal{O}(\log n)$ edges.

Proposition 2.3. Reach *on undirected graphs is in* DynFO$(+, \times)$ *under single-edge deletions and insertions of* $\mathcal{O}(\log n)$ *edges, where* n *is the number of nodes of the graph.*

We use the spanning forest approach as presented in Sketch 2, and re-use its maintenance rules after single-edge deletions. We describe how the auxiliary relations can be updated after $\log n$ many edges are inserted into a graph G. A corresponding update rule may use first-order quantification on the graph and set quantification over a subgraph of size $\mathcal{O}(\log n)$. As sketched above, this MSO quantification can actually be simulated by first-order update rules.

As Reach is MSO-expressible, an update rule can express for each pair a, b of nodes that are affected by the change (i.e. that are adjacent to a new edge after the change) whether they are connected via already existing paths and newly inserted edges. The spanning forest is then updated as follows: a newly inserted edge (a, b) becomes a spanning forest edge if a and b are not connected in the graph which consists of all previously existing edges and all new edges that are lexicographically smaller than (a, b). The in-betweenness relation can be updated similarly in a straightforward fashion, since a node b is between a and c in the new spanning forest if a and b as well as b and c are connected, but a and c become disconnected without b.

As a matter of fact, reachability on *directed graphs* can also be maintained under such insertion operation, at least in the absence of deletions.

The previous example naturally leads to the question of which sizes of bulk changes can be handled by a dynamic program for reachability. It turns out that the reachability query cannot be maintained under changes of more than polylogarithmically many edges.

Sketch 5:

An impossibility result for bulk changes

It turns out that classical lower bound results for the size of AC^0-circuits almost immediately yield upper bounds for the sizes of bulk changes that can be handled. Here, an AC^0-circuit for an input of size n may use polynomially many $\vee$-, $\wedge$-, and $\neg$-gates (with possibly unbounded fan-in) arranged in a circuit of constant depth.

The idea is simple. A classical result by Smolensky states that for computing the parity of the number of ones occurring in a bit string of length n, an AC^0 circuit of depth d requires $2^{\Omega(n^{1/2d})}$ many gates (see [27, Theorem 12.27] for a modern exposition). A simple, well-known reduction yields that deciding reachability for graphs with n edges which are disjoint unions of (undirected) paths also requires AC^0 circuits of size $2^{\Omega(n^{1/2d})}$. Indeed, computing the parity of the number of ones in $w = a_1 \cdots a_n$ can be reduced to

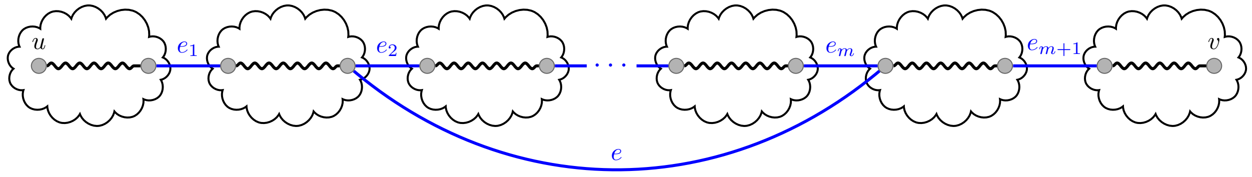

Figure 3: Illustration of the observation that if, after a first-order defined insertion, a path in a graph uses many new edges $e_1, \ldots, e_{m+1}$, there must exist a shortcut via a new edge e with fewer new edges.

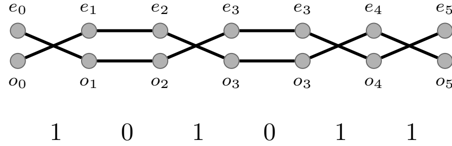

Figure 4: Illustration of the reduction from parity to reachability for the string $w = 101011$.

reachability as follows. A graph G for w can be constructed by converting each bit a_i into a small widget W_i with nodes $e_{i-1}, o_{i-1}, e_i, o_i$ as follows:

- if $a_i = 0$ then $W_i =$
- if $a_i = 1$ then $W_i =$

Now, for a bit string $w = a_1 \ldots a_n$ there is a path from e_0 to e_n iff w has an even number of ones, see Figure 4 for an example. Furthermore, graphs obtained in this fashion are disjoint unions of two paths.

This lower bound translates into a lower bound for first-order formulas via the correspondence of AC^0 and first-order logic due to [2].

Theorem 2.4. *Let $f(n) \in \log^{\omega(1)} n$ be a function from $\mathbb{N}$ to $\mathbb{N}$. There is no first-order formula with access to built-in relations that defines reachability in graphs with at most $f(n)$ edges, even for disjoint unions of (undirected) paths.*

Since from any formula that updates the result of a query after an insertion of $f(n)$ tuples into an initially empty input relation one can construct a formula that defines the query for inputs of size $f(n)$, the following corollary is immediate [9, Corollary 2].

Corollary 2.5. *Let $f(n) \in \log^{\omega(1)} n$ be a function from $\mathbb{N}$ to $\mathbb{N}$. Then reachability (even in disjoint unions of (undirected) paths) cannot be maintained in* DynFO *for bulk changes of size up to $f(n)$, even if the auxiliary relations may be initialised arbitrarily.*

We have seen how to maintain reachability under $\mathcal{O}(\log n)$ edge insertions, and that dynamic first-order programs cannot maintain reachability under insertions of more than a polylogarithmic number of edges. For reachability on undirected graphs this gap can be closed: this query can be maintained under insertions and deletions of polylogarithmic size.

Sketch 6:

Polylog-size changes to undirected graphs

To maintain reachability in undirected graphs under edge changes of polylogarithmic size, the technique of Sketch 4 can be extended. There, we used simulations of MSO formulas on subgraphs of logarithmic size. We do not know whether such simulations are also possible for subgraphs of polylogarithmic size, but we observe next that on subgraphs of this size, NL-computations can be simulated.

First of all, it can be observed that Reach over subgraphs of polylogarithmic size can be expressed by a first-order formula over the whole graph. This follows from the well-known result (see for example [5, p. 613]) that for every $d \in \mathbb{N}$ there is a uniform circuit family for computing the transitive closure of a graph with N nodes using circuits of depth $2d$ and size $N^{\mathcal{O}(N^{1/d})}$. If the subgraph in question has $N \stackrel{\text{def}}{=} \log^c n$ nodes, for some $c \in \mathbb{N}$, we can choose $d \stackrel{\text{def}}{=} 2c$, and the circuit size

$$\begin{aligned} N^{\mathcal{O}(N^{1/d})} &= (\log^c n)^{\mathcal{O}((\log^c n)^{1/d})} = (\log n)^{\mathcal{O}((\log n)^{c/2c})} \\ &= 2^{\mathcal{O}(\log\log n \sqrt{\log n})} \subseteq 2^{\mathcal{O}(\log n)} = n^{\mathcal{O}(1)} \end{aligned}$$

is polynomial in n. This uniform AC^0 circuit family computing reachability for subgraphs of size $\log^c n$ can be turned into an FO(+,×)-formula thanks to [2]. Since Reach is complete for NL under first-order reductions, see [26, Theorem 3.16], first-order logic can thus express all NL-computable queries on graphs of polylogarithmic size.

Now we can sketch the proof idea of the following result [9, Theorem 6].

Theorem 2.6. Reach *on undirected graphs is in* DynFO(+, ×) *under insertions and deletions of*

$\log^c n$ *many edges, for every fixed* $c \in \mathbb{N}$. *Here,* n *is the number of nodes of the graph.*

Again, we employ the spanning forest approach from Sketch 2. When a polylogarithmic number of edges is inserted into a graph, the update rule defines a spanning forest on the at most polylogarithmic number of connected components that get connected by this change, which is possible in first-order logic by NL-simulation as explained above. For each edge in this spanning forest, the lexicographically smallest edge between corresponding components is selected to become part of the spanning forest of the whole graph. The in-between relation is updated accordingly by combining the auxiliary information with in-betweenness information for the spanning forest on the connected components, which again can be expressed directly in first-order logic.

The update after a deletion of polylogarithmically many edges is not much harder. In a first step, the edges are deleted from the spanning forest, and its in-between information is adjusted. Only a polylogarithmic number of connected components of the spanning forest are affected by this step. For them, the update rule checks in a second step whether they can be re-connected by existing non-spanning-tree edges of the graph. This step works exactly as the update for edge insertions.

Just as for Sketch 4, Reach on directed graphs can be maintained under insertions of polylogarithmically many edges, with similar techniques (but in the absence of edge deletions).

2.2 Current frontiers of directed reachability

Turning to directed graphs, we first give a glimpse of an idea how reachability can be maintained under single edge insertions and deletions.

Sketch 7:

Single-edge insertions and deletions in directed graphs

The long-standing question [7, 14, 12, 23, 25, 35, 45] whether reachability on general directed graphs is in DynFO was settled in [8].

Theorem 2.7. Reach *is in* DynFO *under insertions and deletions of single edges.*

The underlying idea is to first reduce the reachability query to a linear-algebraic problem, and then to show that this problem can be maintained with first-order update rules. This approach works if DynFO is closed under the applied reductions it uses, which is guaranteed if they obey two conditions: that they are definable in first-order logic and that one change in the source structure only induces $\mathcal{O}(1)$ changes in the target structure. Such *bounded first-order (bfo) reductions* were introduced in [35].

Step 1: Reduction to FullRank.

Problem:	FullRank
Input:	An $n \times n$-matrix C
Question:	Is the rank of C equal to n?

This step is very similar to reductions used by Cook (for studying the NC-hierarchy) and Laubner (for studying extensions of first-order logic by linear-algebraic operators) [6, 30]. To facilitate subsequent generalisations, we describe the reduction to FullRank by two reductions with another intermediate problem. We defer to [8] for further details.

Suppose that A is the adjacency matrix of a graph G. The number of paths of length i from s to t in G corresponds to the value of the s-t-entry of A^i, the i-th power of the adjacency matrix. The matrix $I - \frac{1}{n}A$ is invertible (since diagonally dominant) and its inverse can be written, analogously to standard geometric series, as:

$$(I - \frac{1}{n}A)^{-1} = \sum_{i=0}^{\infty} \frac{1}{n^i} A^i$$

Hence, there is a path from s to t if the s-t-entry of the inverse of $C \stackrel{\text{def}}{=} I - \frac{1}{n}A$ is not zero. This yields a bfo-reduction from Reach to the problem MatrixInverse$^{\neq 0}$, which we define as:

Problem:	MatrixInverse$^{\neq 0}$
Input:	Invertible $n \times n$-matrix C; $s, t \leq n$
Question:	Is the s-t-entry of C^{-1} not 0?

The problem MatrixInverse$^{\neq 0}$ can then be reduced to FullRank: by Cramer's rule, an entry of a matrix C^{-1} is non-zero if and only if the determinant of some submatrix of C is non-zero, which is equivalent to the question whether this submatrix has full rank. We refer to [8] for details and for a verification that the reductions are actual bfo-reductions.

Step 2: Maintaining FullRank. Now our goal is to update whether a matrix C has full rank under changes of one entry. This can be done similarly as described in [19]: we maintain matrices B, E such that $BC = E$, B is invertible, and E is in reduced row-echelon form. It turns out that, modulo small primes, the matrices B and E can be updated using a constant number of simple matrix operations under changes of single entries of C. Reachability can then ultimately be maintained by maintaining the full rank property for a suitable number of such small primes.

This result has since been simplified and improved. The following technique for reducing the conceptual re-

quirements for maintaining a query has been useful for this purpose, and it has also been applied to show a number of other maintenance results.

Sketch 8:

The muddling technique

The *muddling technique* exploits that under certain conditions it suffices to maintain the result of a query for polylogarithmically many change steps, as opposed to arbitrarily many. In the following, we only consider queries that are *domain independent* in the sense that the query result does not change for an instance when additional, isolated elements are added to the domain. A query is $(\text{NL}, f(n))$*-maintainable*, if it can be maintained for $f(n)$ change steps starting from an arbitrary database instance and auxiliary data initialised by an NL computation.

THEOREM 2.8 (MUDDLING LEMMA [10, 11, 40]). *Let $\mathcal{Q}$ be a domain independent query that is $(\text{NL}, \log n)$-maintainable under some set Δ of change operations.*[8]

a) If Δ is a set of single-tuple change operations then $(\mathcal{Q}, \Delta)$ is in DYNFO.

b) If $\mathcal{Q} \in \text{NL}$ and Δ is a set of bulk change operations of size at most $\log^d n$, for an arbitrary $d \in \mathbb{N}$, then $(\mathcal{Q}, \Delta)$ is in DYNFO$(+, \times)$.

As an example, the Muddling Lemma allows to prove that a query is in DYNFO by showing that it can be maintained for $\log n$ many steps, starting from an arbitrary graph G with n nodes, with the help of auxiliary relations that can be obtained from G by some NL computation.

A result that highlights the power of this technique is that all queries expressible in monadic second order logic are in DYNFO under changes of single tuples, if the database (always) has bounded treewidth [10]. We do not know how to maintain a tree decomposition with first-order formulas, yet the muddling lemma allows to pre-compute a tree decomposition in LOGSPACE. It can be shown that a query result can then be maintained for $\log n$ steps.[9]

For the reachability query on undirected graphs, the maintenance strategy could be lifted from single-edge changes to changes of polylogarithmic size. It is an immediate question to what extent this is possible for directed graphs.

We recall that one can maintain REACH under insertions of polylogarithmic size with the techniques of Sketch 6, but this result does not allow for any edge deletions. It turns out that bulk insertions *and* deletions are indeed possible, but the allowed number of changed edges so far falls short of $\log n$.

Sketch 9:

Almost $\log n$ insertions and deletions into directed graphs

We now show how the approach of Sketch 7 can be adapted such that, using the muddling technique, reachability on directed graphs can be maintained under a non-constant number of edge insertions and deletions [11, Theorem 1].

THEOREM 2.9. REACH *is in* DYNFO$(+, \times)$ *under insertions and deletions of edges that affect $\mathcal{O}(\frac{\log n}{\log \log n})$ nodes, on graphs with n nodes.*

In Sketch 7 we explained how REACH can be reduced to MATRIXINVERSE$^{\neq 0}$ and to FULLRANK, and how to maintain the latter. Here, the idea is to maintain MATRIXINVERSE$^{\neq 0}$ directly, by maintaining (sufficient information on) the inverse C^{-1} of the input matrix C. Therefore, our goal is to update the inverse C^{-1} when C changes to some matrix $C + \Delta C$.

Suppose that ΔC is a change matrix that encodes edge insertions and deletions that affect $k \stackrel{\text{def}}{=} \frac{\log n}{\log \log n}$ nodes of the graph. Then, ΔC has at most k non-zero rows and columns, and can be written as a matrix product $\Delta C = UBV$ where B has dimension $k \times k$. The update of C^{-1} to $(C + \Delta C)^{-1}$ with $\Delta C = UBV$ is described by the Sherman-Morrison-Woodbury identity (cf. [24]) as

$$\begin{aligned}(C + \Delta C)^{-1} &= (C + UBV)^{-1} \\ &= C^{-1} - C^{-1}U(I + BVC^{-1}U)^{-1}BVC^{-1}.\end{aligned}$$

To implement the right-hand-side of this identity as a dynamic program with first-order update rules, some obstacles have to be eliminated. First, literally computing the identity is not possible in first-order logic, since entries in C^{-1} can be exponentially large, and multiplying such numbers is not possible with first-order formulas even in the presence of arithmetic on the domain. A workaround is to compute C^{-1} modulo polynomially many, polynomially bounded primes: an entry of C^{-1} is non-zero if and only if it is non-zero modulo one of the primes.

Since $I + BVC^{-1}U$ is a $k \times k$ matrix, its inverse can be computed in AC^0 over $\mathbb{Z}_p$, for every prime p that is polynomially bounded in n — if it is invertible. However, although the occurring matrices are

[8] The result actually even holds for $(\text{AC}^c, \log^c n)$-maintainable queries, for an arbitrary $c \in \mathbb{N}$, and the proof uses the fact that AC^c-circuits correspond to fixed-point computations with $\mathcal{O}(\log^c n)$ iterations, cf. Section 5 in [26].

[9] In fact, for the MSO result an *annotated* tree decomposition is needed and therefore a stronger version of the Muddling Lemma is used.

all invertible over $\mathbb{Q}$, they may not be invertible over $\mathbb{Z}_p$ for some primes p. If this is the case for a prime, the auxiliary relations for this prime become invalid and cannot be used any more. But thanks to the muddling technique it suffices to maintain the query for a polylogarithmic number of change steps, and it is possible to guarantee that a sufficiently large number of primes survives for that many rounds, to get the final result.

It is an open question whether reachability on directed graphs can be maintained under insertions and deletions of logarithmically or even poly-log many edges using first-order update rules. By allowing update rules from stronger logics than first-order logic, this becomes possible: with additional majority quantifiers one can maintain reachability on all directed graphs under changes of poly-log size [11]; for certain classes of directed graphs, additional parity quantifiers are sufficient [9].

Sketch 10:

Regular path queries

Attentive readers might have observed a gap in our reasoning, as presented so far: our motivating scenario involved graph databases and regular path queries but throughout this article, we studied mere reachability queries on graphs without edge labels. However, it turns out that the maintainability of the latter is actually the key for maintaining regular path queries (and then conjunctive regular path queries and unions therefore, and so on). This is because the evaluation of a regular path query can be reduced to the reachability query in a very simple fashion [28].

Indeed, this is doable by considering the product of the actual graph with an automaton for a regular language. More precisely, if A is an NFA that decides the regular language R underlying the regular path query at hand, and if D is a graph database with edges labelled by the alphabet used by A, then the question whether there is an R-path from u to v boils down to the question whether the node (s_f, v) is reachable from the node (s_0, u) in the synchronised product $A \times D$. The nodes of that product are pairs (s, w) of a state of A and a node from D and there is an edge from (s_1, w_1) to (s_2, w_2) if, for some symbol a, there is an a-transition from s_1 to s_2 in A and an a-labelled edge from w_1 to w_2 in D. Furthermore, s_0 and s_f are the unique initial and final states of A, respectively.

This reduction is actually a bounded-first order reduction, since each single change in D only induces at most size(A) many, first-order definable, changes in $D \times A$. Therefore, maintainability of Reach on directed graphs yields maintainability of the R-path query on graph databases, for every R. As an example, we get the following corollary from Theorem 2.9.

Corollary 2.10. *Let $\mathcal{Q}$ be a regular path query. Then $\mathcal{Q}$ is in* DynFO$(+, \times)$ *under insertions and deletions of edges that affect $\mathcal{O}(\frac{\log n}{\log \log n})$ nodes, where n is the number of nodes of the graph database.*

3. QUERY MAINTENANCE BARRIERS

First-order update rules are surprisingly powerful. Above, we explored the reachability query and saw that it can be updated with such rules, even in cases, where complex changes are allowed. Also the tree isomorphism query [17], all MSO queries on bounded tree-width graphs [10], and all context-free languages [22] can be maintained in DynFO.

This leads to the natural question: Is there a barrier for the power of dynamic programs, besides the easy observation that all queries in DynFO are computable in polynomial time? Proving such barriers is a challenging task already in static settings, and it is therefore not surprising that so far there are only preliminary answers. Much of the work on barriers for dynamic programs was done in the quest of finding out whether reachability is in DynFO. For this reason we focus on results that establish barriers for updating reachability in scenarios with restricted resources such as small auxiliary data and restricted updated rules.

While we can rely on several methods for proving barriers of inexpressibility for first-order logic in static scenarios, our tool set for dynamic lower bounds is much less developed. Classical methods for static inexpressibility include Ehrenfeucht-Fraïssé games and locality-based arguments [16, 31] as well as circuit-based methods [27] that exploit the connection between first-order logic and constant-depth circuits. Parity (of a unary relation) and reachability are standard examples for queries, that are provably not expressible in first-order logic. Yet, both queries are contained in DynFO.

In the following, we outline two tools for dynamic lower bounds: (a) exploitation of static lower bounds and (b) a locality method for restricted update rules.

3.1 Exploiting static methods

Many non-maintainability results for DynFO were shown by contradiction with the help of known static lower bounds. More precisely, it was shown that if there was a dynamic program for a particular query $\mathcal{Q}_{\text{dynamic}}$, then some $\mathcal{Q}_{\text{static}}$ would be expressible in first-order logic, maybe in the presence of "helpful relations", contradicting known inexpressibility results.

After making the notion of *helpful relations* precise, we present two instantiations of this technique which were used to establish that queries cannot be maintained by first-order updates when the arity – and therefore the size – of auxiliary relations is restricted.

A query over schema τ is *definable with helpful relations* over schema τ_{help} if there is a formula φ over $\tau \cup \tau_{\text{help}}$ such that for each database $\mathcal{D}$ over τ there is a database $\mathcal{D}_{\text{help}}$ over τ_{help} such that evaluating φ on $(\mathcal{D}, \mathcal{D}_{\text{help}})$ yields the result of the query on $\mathcal{D}$.

A first set of non-maintainability results for DynFO with unary auxiliary relations can be derived from inexpressibility results for existential, monadic second-order logic EMSO. This logic extends first-order logic by existential quantification of unary relations. Roughly speaking, inexpressibility results for EMSO often transfer to inexpressibility results for first-order logic with unary helpful relations, which in turn allow proving barriers for DynFO with unary auxiliary relations.

Sketch 11:

Unary auxiliary relations do not suffice for Reach

EMSO formulas can not define the transitive closure of simple paths (e.g., implicitly in [18]). By basically the same arguments as in [18] it can be shown that, for each first-order formula $\varphi(x, y)$ and each large enough n, there is no tuple H of help relations, such that φ defines the transitive closure over E on (G, H), where G is just a simple path [14, Theorem 4.3]. This "static" inexpressibility result implies the following "dynamic" inexpressibility result.

Theorem 3.1 ([14]). Reach *is not in* DynFO *with only unary auxiliary relations and one binary auxiliary relation for storing the transitive closure.*

Towards a contradiction, suppose Reach can be maintained in DynFO under edge deletions with unary auxiliary relations in addition to the binary relation T for the transitive closure. Then the first-order update rule for edge deletions can be used to construct a formula φ that defines the transitive closure on simple paths, using unary help relations. Indeed, let G be a simple path on nodes $1, \ldots, n$. The help relations H can simply be chosen as the unary auxiliary relations used by the dynamic program for the cycle C that extends E by the edge $(n, 1)$. The formula φ results from the update rule for deletions, by replacing every atom $T(x, y)$ by $\top$, since the transitive closure relation of a cycle is the full binary relation. This yields the desired contradiction.

We next give an example for deriving dynamic inexpressibility results from circuit lower bounds.

Sketch 12:

An arity hierarchy for auxiliary relations

It is well-known that the parity of n bits cannot be computed by constant-depth circuits of polynomial size [1, 21]. A less known result by Cai [4] extends this to the presence of "helpful bits": given n bit strings of length n^6, a constant-depth circuit of polynomial size cannot compute the parity of each of these strings even with $n-1$ help bits (which may depend on the bit stings). This result translates to first-order logic where, roughly speaking, the help bits translate to helpful relations: there is a query over a $6k$-ary schema which cannot be expressed by a first-order formula with $(k-1)$-ary help relations, for all $k \in \mathbb{N}$. Again, a barrier for DynFO follows immediately.

Theorem 3.2 ([14, 15]). *Let $k \geq 2$. There is a query over a $(3k+1)$-ary schema which can be maintained in* DynFO *with k-ary auxiliary relations, but not with $(k-1)$-ary auxiliary relations. In particular,* DynFO *has a strict arity hierarchy.*

It is open whether DynFO has a strict arity hierarchy over a fixed schema.[10]

3.2 Locality methods for restricted update rules

The above techniques work for first-order update rules, yet only for restricted arities: for queries on graphs, we currently only know how to prove barriers with respect to unary auxiliary relations. We now present a technique that allows proving barriers for high-arity auxiliary relations, yet it can only be applied to quantifier-free update rules and slight extensions thereof.

Quantifier-free update rules might seem unreasonably weak, but it turns out that they are not entirely powerless. As an example, in Sketch 1 we showed how reachability on directed graphs can be maintained under edge insertions with quantifier-free update rules. Also, membership of strings in regular language can be maintained without quantifiers [22].[11]

The *Substructure Method* encapsulates the weakness of quantifier-free update rules as a technical lemma. We sketch it next and give three applications.

Sketch 13:

Barriers with the Substructure Method

The intuition of the Substructure Method is as follows: suppose $\mathcal{S}$ is the current state of a dynamic program, i.e., $\mathcal{S}$ is a structure consisting of the input database and the auxiliary database. When updating an auxiliary tuple $\vec{c}$ after modifying a tuple $\vec{d}$, a quantifier-free update rule only has access to $\vec{c}$ and $\vec{d}$.[12] Thus, if a sequence of modifications changes only tuples from a substructure $\mathcal{A}$ of $\mathcal{S}$, then the auxiliary

[10] Such a strict hierarchy has been established for update rules without quantifiers [43, 41].

[11] In fact, the regular languages can be *characterised* by this property.

data of $\mathcal{A}$ is not affected by information outside $\mathcal{A}$. In particular, two isomorphic substructures $\mathcal{A}$ and $\mathcal{B}$ remain isomorphic, when corresponding modifications are applied to them.

LEMMA 3.3 (SUBSTRUCTURE LEMMA [22, 44]). *Let $\mathcal{P}$ be a dynamic program with quantifier-free update rules and let $\mathcal{S}$ and $\mathcal{T}$ be states of $\mathcal{P}$ with isomorphic substructures $\mathcal{A}$ and $\mathcal{B}$, respectively. Then the substructures $\mathcal{A}$ and $\mathcal{B}$ are still isomorphic after applying isomorphism-respecting changes α to $\mathcal{A}$ and β to $\mathcal{B}$. In particular, if $\mathcal{P}$ has a Boolean relation Q for storing a query result, then Q has the same value in the resulting states.*

Now, to prove that a query $\mathcal{Q}$ cannot be maintained with quantifier-free update rules using the Substructure Method, one can proceed as follows. Assume, towards a contradiction, that there is a program for $\mathcal{Q}$. Then, find two states $\mathcal{S}$ and $\mathcal{T}$ of a dynamic program with two isomorphic substructures $\mathcal{A}$ and $\mathcal{B}$, respectively, such that applying two corresponding modification sequences α and β to $\mathcal{A}$ and $\mathcal{B}$ yields one structure $\mathcal{S}'$ in $\mathcal{Q}$ and one structure $\mathcal{T}'$ not in $\mathcal{Q}$. By the Substructure Lemma, this is a contradiction.

The challenge is to find suitable structures $\mathcal{S}$ and $\mathcal{T}$ for a query at hand. Several combinatorial techniques have been used for finding such structures for which we provide examples. The proof of the following result combines the Substructure Method with a simple counting argument [22, Proposition 6.2].

THEOREM 3.4. *Alternating reachability cannot be maintained with quantifier-free update rules.*

By combining Ramsey's Theorem and Higman's Lemma to find suitable structures with isomorphic substructures, a barrier for REACH can be shown, though only for restricted auxiliary relations [45, Theorem 4.7].

THEOREM 3.5. REACH *cannot be maintained with quantifier-free update rules and binary auxiliary relations.*

It is open whether REACH can be maintained with quantifier-free update rules under single edge modifications. However, combining the Substructure Lemma with upper and lower for Ramsey numbers, a technique introduced in [43], it can be shown that REACH cannot be maintained without quantifiers under moderate definable changes [39, Theorem 7.3].

THEOREM 3.6. REACH *cannot be maintained with quantifier-free update rules under changes defined by quantifier-free first-order formulas.*

4. SUMMARY AND FURTHER WORK

We have presented several DYNFO maintainability results for the reachability query, along with the techniques that are used to construct the corresponding dynamic programs. As discussed in Sketch 10, these results can readily be translated into maintenance results for regular path queries. Further DYNFO maintainability results, also for extensions of regular path queries, are given in [42, 32, 3].

Of course, the *lower bounds* from Section 3 directly hold for regular path queries. Apart from the barriers discussed there, some further challenges become visible when trying to construct dynamic programs for graph database queries.

We emphasise that the maintenance results for REACH do not imply that all NL-computable queries are in DYNFO, although REACH is NL-complete. This is because DYNFO is only (known to be) closed under *bounded* first-order reductions and REACH is provably not NL-complete under these reductions [35].

However, relatively easy graph queries can be shown to be NL-complete under bfo reductions, as for example the $(a[bc])^*$ *query* that selects all pairs (u, v) of nodes such that there is an a-labelled path from u to v, and every intermediate node on this path is the start of a path of length 2 labelled bc. This query can be defined via *nested regular expressions* [36], instead of regular expressions used to define regular path queries.

If the $(a[bc])^*$ query is shown to be in DYNFO under single-edge changes, then so are all NL queries. As there are queries with much smaller static complexity than NL which are not known to be in DYNFO [41], such a result seems unlikely. It is not even known whether the $(a[bc])^*$ query can be maintained when only insertions of single edges are allowed.

On a more technical note, we remark that the settings of [34] and [13] are slightly different, in that the latter allows to change the set of nodes of the graph. It turns out that this difference hardly matters for single-tuple changes and only mildly for more complex changes. Maintainability of queries usually coincides in both settings. We stuck here to the setting of [34], mainly because of its simplicity.

How to approach query maintenance in DynFO?

We have seen some queries that can be maintained in DYNFO and others where this question is open. How should one try to find out whether a given query Q is in DYNFO? Although there is no truly systematic approach to showing that a given query is in DYNFO, the following guiding questions can serve as a heuristic on how to start.

[12] In general, the database could also have some constants. But we assume here, that it does not, for simplicity.

(a) Is the static complexity of Q above NC?

  - If this is the case, e.g. if Q is P-hard, the chances of successfully maintaining it are low.[13]

(b) Is the query hard under bfo reductions for some class $\mathcal{C}$ above AC^0, e.g. NL or LOGSPACE?

  - If that is the case, it will likely still be difficult to show that Q is in DynFO, since that would imply that all queries from $\mathcal{C}$ are in DynFO.

(c) Otherwise, the methods described in this article might be successful. Probably it does not hurt to try the muddling technique first.

Future work

Some open questions that might be worth tackling are the following.

Open question 1. *Can reachability on directed graphs be maintained with first-order update rules under changes of polylogarithmic size?*

Open question 2. *Can minimal distances and witness paths between nodes of a graph be maintained with first-order update rules?*

Open question 3. *Can reachability be maintained with quantifier-free update formulas under single-edge deletions?*

Besides maintenance of reachability and related queries, other aspects of dynamic complexity have been studied as well. These include static analysis of dynamic programs [38] as well as connections to information extraction [20] and parameterised complexity [37].

Another exciting research question is to bridge the gap between dynamic complexity and dynamic algorithms: most of the above results are pure expressibility results and the dynamic programs are not very efficient with respect to their overall work. In future work we plan to investigate under which circumstances queries can be maintained in a work-efficient fashion.

[13]However, there are some rather artificial P-complete problems that are in DynFO [35].

5. REFERENCES

[1] M. Ajtai. Σ^1_1 formulae on finite structures. *Ann. of Pure and Applied Logic*, 24:1–48, 1983.

[2] D. A. M. Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *J. Comput. Syst. Sci.*, 41(3):274–306, 1990.

[3] P. Bouyer and V. Jugé. Dynamic complexity of the Dyck reachability. In *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017*, pages 265–280, 2017.

[4] J. Cai. Lower bounds for constant-depth circuits in the presence of help bits. *Inf. Process. Lett.*, 36(2):79–83, 1990.

[5] X. Chen, I. C. Oliveira, R. A. Servedio, and L. Tan. Near-optimal small-depth lower bounds for small distance connectivity. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016*, pages 612–625. ACM, 2016.

[6] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1-3):2–21, 1985.

[7] S. Datta, W. Hesse, and R. Kulkarni. Dynamic complexity of directed reachability and other problems. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Part I*, pages 356–367, 2014.

[8] S. Datta, R. Kulkarni, A. Mukherjee, T. Schwentick, and T. Zeume. Reachability is in DynFO. *J. ACM*, 65(5):33:1–33:24, 2018.

[9] S. Datta, P. Kumar, A. Mukherjee, A. Tawari, N. Vortmeier, and T. Zeume. Dynamic complexity of reachability: How many changes can we handle? In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020*, pages 122:1–122:19, 2020.

[10] S. Datta, A. Mukherjee, T. Schwentick, N. Vortmeier, and T. Zeume. A strategy for dynamic programs: Start over and muddle through. *Logical Methods in Computer Science*, 15(2), 2019.

[11] S. Datta, A. Mukherjee, N. Vortmeier, and T. Zeume. Reachability and distances under multiple changes. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, pages 120:1–120:14, 2018.

[12] G. Dong, L. Libkin, J. Su, and L. Wong. Maintaining transitive closure of graphs in SQL. *International Journal of Information Technology*, 51(1):46, 1999.

[13] G. Dong and J. Su. First-order incremental evaluation of datalog queries. In *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages*, pages 295–308, 1993.

[14] G. Dong and J. Su. Arity bounds in first-order incremental evaluation and definition of polynomial time database queries. *J. Comput. Syst. Sci.*, 57(3):289–308, 1998.

[15] G. Dong and L. Zhang. Separating auxiliary arity hierarchy of first-order incremental evaluation systems using (3k+1)-ary input relations. *Int. J. Found. Comput. Sci.*, 11(4):573–578, 2000.

[16] H.-D. Ebbinghaus and J. Flum. *Finite model theory.* Springer Science & Business Media, 2005.
[17] K. Etessami. Dynamic tree isomorphism via first-order updates. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1998*, pages 235–243, 1998.
[18] R. Fagin. Monadic generalized spectra. *Math. Log. Q.*, 21(1):89–96, 1975.
[19] G. S. Frandsen and P. F. Frandsen. Dynamic matrix rank. *Theor. Comput. Sci.*, 410(41):4085–4093, 2009.
[20] D. D. Freydenberger and S. M. Thompson. Dynamic complexity of document spanners. In *23rd International Conference on Database Theory, ICDT 2020*, pages 11:1–11:21, 2020.
[21] M. L. Furst, J. B. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.
[22] W. Gelade, M. Marquardt, and T. Schwentick. The dynamic complexity of formal languages. *ACM Trans. Comput. Log.*, 13(3):19, 2012.
[23] E. Grädel and S. Siebertz. Dynamic definability. In *15th International Conference on Database Theory, ICDT 2012*, pages 236–248, 2012.
[24] H. V. Henderson and S. R. Searle. On deriving the inverse of a sum of matrices. *Siam Review*, 23(1):53–60, 1981.
[25] W. Hesse. The dynamic complexity of transitive closure is in DynTC^0. *Theor. Comput. Sci.*, 296(3):473–485, 2003.
[26] N. Immerman. *Descriptive Complexity.* Springer, 1999.
[27] S. Jukna. *Boolean function complexity: advances and frontiers*, volume 27. Springer Science & Business Media, 2012.
[28] D. Kähler and T. Wilke. Program complexity of dynamic LTL model checking. In *Computer Science Logic, CSL 2003*, pages 271–284, 2003.
[29] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.
[30] B. Laubner. *The structure of graphs and new logics for the characterization of Polynomial Time.* PhD thesis, Humboldt University of Berlin, 2011.
[31] L. Libkin. *Elements of Finite Model Theory.* Springer, 2004.
[32] P. Muñoz, N. Vortmeier, and T. Zeume. Dynamic graph queries. In *19th International Conference on Database Theory, ICDT 2016*, pages 14:1–14:18, 2016.
[33] M. Nikolic and D. Olteanu. Incremental view maintenance with triple lock factorization benefits. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018*, pages 365–380. ACM, 2018.
[34] S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1994*, pages 210–221, 1994.
[35] S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. *J. Comput. Syst. Sci.*, 55(2):199–209, 1997.
[36] J. Pérez, M. Arenas, and C. Gutiérrez. nSPARQL: A navigational language for RDF. *J. Web Semant.*, 8(4):255–270, 2010.
[37] J. Schmidt, T. Schwentick, N. Vortmeier, T. Zeume, and I. Kokkinis. Dynamic complexity meets parameterised algorithms. In *28th EACSL Annual Conference on Computer Science Logic, CSL 2020*, pages 36:1–36:17, 2020.
[38] T. Schwentick, N. Vortmeier, and T. Zeume. Static analysis for logic-based dynamic programs. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015*, pages 308–324, 2015.
[39] T. Schwentick, N. Vortmeier, and T. Zeume. Dynamic complexity under definable changes. *ACM Trans. Database Syst.*, 43(3):12:1–12:38, 2018.
[40] N. Vortmeier. *Dynamic expressibility under complex changes.* PhD thesis, TU Dortmund University, Germany, 2019.
[41] N. Vortmeier and T. Zeume. Dynamic complexity of parity exists queries. In *28th EACSL Annual Conference on Computer Science Logic, CSL 2020*, pages 37:1–37:16, 2020.
[42] V. Weber and T. Schwentick. Dynamic complexity theory revisited. *Theory Comput. Syst.*, 40(4):355–377, 2007.
[43] T. Zeume. *Small Dynamic Complexity Classes: An Investigation into Dynamic Descriptive Complexity*, volume 10110 of *Lecture Notes in Computer Science*. Springer, 2017.
[44] T. Zeume and T. Schwentick. Dynamic conjunctive queries. In *Proc. 17th International Conference on Database Theory (ICDT 2014)*, pages 38–49, 2014.
[45] T. Zeume and T. Schwentick. On the quantifier-free dynamic complexity of reachability. *Inf. Comput.*, 240:108–129, 2015.