Dynamic Conjunctive Queries

Thomas Zeume
TU Dortmund University
thomas.zeume@cs.tu-dortmund.de

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

1. INTRODUCTION

The re-evaluation of a fixed query after an update to a 
huge database can be a time-consuming process; in particular 
when it is performed from scratch. For this reason previ-
ously computed information such as the old query result 
and (possibly) other auxiliary information is often reused 
in order to speed up the process.

This maintenance of query results has attracted lots of 
attention over the last decades of database related research. 
For relational databases an algorithmic approach (see e.g. 
[17, 11]) and a declarative approach (see e.g. [7, 10]) have 
been studied. Here, we continue the study of the declarative 
approach where query results are updated by queries from

some query language.

One possible formalization of this approach is the descrip-
tive dynamic complexity framework (short: dynamic com-
plexity) introduced by Patnaik and Immerman [16]. For a 
relational database subject to change, auxiliary relations are 
maintained with the intention to help answering a query \( Q \). 
When an update to the database, an insertion or deletion of 
a tuple, occurs, every auxiliary relation is updated through 
a first-order query that can refer to both, the database and 
the auxiliary relations. A particular auxiliary relation shall 
always represent the answer to \( Q \). The class of all queries 
maintainable in this way, and thus also in the core of SQL, 
is called DynFO.

The class DynFO is quite powerful. Many queries in-
expressible in (static) first-order logic, such as the transi-
tive closure query on undirected graphs [16] and the word 
problem for context-free languages [9], can be maintained 
in DynFO. In [8] a query inexpressible in first-order logic 
extended by a transitive closure operator and counting has 
been shown to be contained in DynFO. There are no general 
inexpressibility results for DynFO at all\(^1\).

Towards a deeper understanding of the dynamic setting, 
two main restrictions of DynFO have been explored in the 
literature. Dong and Su started the study of restricted 
auxiliary relations [6]. They restricted the arity of auxil-
iary relations and obtained inexpressibility results for unary 
auxiliary relations. On the other hand, Hesse started the 
exploration of syntactic fragments of DynFO, such as the 
one obtained by disallowing quantification in update formu-
las [14]. Inexpressibility results for this particular fragment 
have been obtained in [9].

In this work we investigate classes of queries maintainable 
by conjunctive queries and extensions thereof, thus we are 
following the approach of Hesse.

Conjunctive queries (CQs), that is, in terms of logic, 
existential first-order queries whose quantifier-free part is 
a conjunction of atoms, are one of the most investigated 
query languages. Starting with Chandra and Merlin [2], 
who analyzed conjunctive queries for relational databases, 
those queries have been studied for almost every emerging 
new database model. Usually also the extension by unions 
(UCQs), by negations (CQ’s) as well as by both unions 
and negations (UCQ’s or, equivalently, 3 FO) have been 
studied. It is folklore that all those classes are distinct for 
relational databases.

In this work we aim at the following goals.

\(^1\)Except for the trivial ones due to the fact that queries main-
tainable in DynFO can be computed in polynomial time.
GOAL 1. Understand the relationship between different classes of dynamic conjunctive queries as well as their relationship to static query classes.

Our focus for relating variants of dynamic conjunctive queries is on the classes listed above. Some preliminary results have been obtained in [19] for the quantifier-free variants of conjunctive queries.

As for the relationship to static classes, it is interesting to understand whether larger static classes $C$ can be captured by a dynamic class $\text{DynC}^\prime$, for some weaker $C'$. Up to now only two such results are known, namely, that MSO can be characterized by quantifier-free DynFO on strings and that $\exists^*\text{FO}$ is captured by quantifier-free DynFO extended by auxiliary functions on general structures [9].

In the framework of Patnaik and Immerman auxiliary relations are always re-defined as a whole after an update. However, in the context of query re-evaluation, it is often convenient to express the new state of an auxiliary relation $R$ in terms of the current relation and some “Delta”, that is, by specifying tuples $R^+$ to be inserted into $R$ and tuples $R^-$ to be removed from $R$. We refer to the former semantics as absolute semantics and to the latter as $\Delta$-semantics. Obviously, the choice of the semantics does not affect the expressiveness of an update language that is closed under Boolean operations. However, most of the update languages in this paper lack some Boolean closure properties.

GOAL 2. Understand the relationship between absolute semantics and $\Delta$-semantics for conjunctive queries and their variants.

In this work we contribute to achieve those two goals as follows.

Contributions. For an overview of the relationship of the various dynamic classes of conjunctive queries we refer to Figure 1.

The distinctness of those classes in static relational databases does not translate into the dynamic setting.\(^2\)

- We show that, in many cases, the addition of the union-operator does not yield additional expressive power in the dynamic setting, for example,DynUCQ\(^\ast\) = DynUCQ\(^\ast\), DynUCQ = DynCQ, and DynPropUCQ\(^\ast\) = DynPropUCQ\(^\ast\), where Prop indicates classes without quantifiers.

- In other cases, negation does not increase the expressive power of an update language, e.g., we have DynPropUCQ = DynPropUCQ and $\Delta$-DynCQ\(^\ast\) = $\Delta$-DynCQ.

- Further, often quantifiers can be replaced by their dual quantifiers, e.g., Dyn$\exists^3$FO(= DynUCQ\(^\ast\)) = Dyn$\forall^3$FO.

Whether DynCQ\(^\ast\) = DynCQ remains open. However, a first-step is taken towards the separation of the remaining fragments:

- We show that dynamic conjunctive queries without negations and quantifiers are strictly weaker than the quantifier-free fragment of DynFO.

Furthermore, we show that dynamic conjunctive queries extended by negations capture all first-order queries:

- We characterize the class of first-order queries as the class maintainable by non-recursive dynamic Dyn$\exists^3$FO-programs with a single existential quantifier per update formula. This implies that dynamic conjunctive queries extended by negations can maintain all first-order queries.

For the second goal, the main finding is that the difference between absolute and $\Delta$-semantics is much smaller than we had expected.

- The dynamic classes corresponding to FO, CQ\(^\ast\) and ProP yield the same expressive power with respect to absolute and $\Delta$-semantics.

- It turns out that conjunctive queries and conjunctive queries with negation coincide with respect to $\Delta$-semantics, that is, in particular, $\Delta$-DynCQ = $\Delta$-DynCQ\(^\ast\) and thus, also $\Delta$-DynCQ = DynUCQ\(^\ast\).

Choice of setting. The concrete settings under which dynamic complexity has been studied in the literature slightly differ in several aspects. We shortly discuss the most important aspects, what choice we took for this work and why we made this choice.

An important aspect is whether to use a finite and fixed domain, an active domain or an infinite domain. In this work, we follow the framework of Patnaik and Immerman in which the domain is finite and fixed [16]. To maintain a query, a dynamic program has to work uniformly for every domain. This fixed domain framework for a dynamic setting might appear counterintuitive at first sight. However, it allows to study the underlying dynamic mechanisms of dynamic programs, in particular when one is interested to develop lower bound methods. Fixed domains also offer a strong connection to logics and circuit complexity. In incremental evaluation systems (IES), a framework proposed by Dong and Topor [7], active domains are used. This setting is a little closer to real database systems but most results in dynamic complexity hold equally in both frameworks.

Another parameter to choose is how the auxiliary data is initialized. In the setting of Patnaik and Immerman, dynamic programs start from empty databases and the auxiliary data is either initialized by a polynomial time computation or by a formula from the same class as the update formulas. Later this was generalized by Weber and the second author by proposing that dynamic programs start from an arbitrary initial database and auxiliary data initialized by a mapping computable in some given complexity class [18].

In this work we allow for arbitrary initialization mappings. This is motivated by our long term goal to develop lower bound techniques for dynamic programs. While lower bounds in settings with restricted initialization might depend on this restriction, an inexpressibility result in the setting with arbitrary initialization, on the other hand, really shows that a query cannot be maintained. A result like DynUCQ = DynCQ is helpful for the development of lower bound techniques, as it shows that for proving lower bounds for DynUCQ it is sufficient to consider DynCQ programs — but also that one has to be aware that lower bounds for

\(^2\)The notation for classes will be formally introduced in Sections 2 and 4. In general, $\Delta$ indicates $\Delta$-semantics, the absence of $\Delta$ indicates absolute semantics.
DynCQ are as hard as lower bounds for DynUCQ. However, though all our results are stated for arbitrary initialization mappings, they also hold in the setting with empty initial database and first-order initialization for the auxiliary data. The proofs do not carry over to the strict setting of Patnaik and Immerman where, in a dynamic class DynC, only C initializations are allowed.

**Related work.** We next discuss some further related work, beyond what we already mentioned above. The expressivity of first-order logic in the dynamic complexity frameworks discussed above has been studied a lot (see e.g. [16, 6, 8, 13, 14, 18, 10, 9]). Most results focus on showing that a problem from some static complexity class can be dynamically maintained by programs of a weaker query class. Some lower bounds have been achieved as well (see e.g. [5, 4, 6, 9, 10, 19]). Many other aspects such as the arity of auxiliary relations (see e.g. [6, 14]), whether the auxiliary relations are determined by the current structure (see e.g. [16, 5, 10]), and the presence of an order (see e.g. [10]) have been studied.

An algebraic perspective of incremental view maintenance under Δ-semantics has been studied in [15]. Parts of this work have also been implemented, see e.g., [1].

**Outline.** In Section 2 we define our dynamic setting more precisely. In Section 3 the results for absolute semantics are presented. The alternative Δ-semantics is introduced and studied in Section 4. In Section 5 we give the dynamic characterization of first-order logic. We conclude with a discussion and a first step towards separations in Section 6.

**Acknowledgement.** We thank Nils Vortmeier for careful proofreading. Further we are grateful to the anonymous reviewers for several very helpful comments. We acknowledge the financial support by the German DFG under grant SCHW 678/6-1.

2. DYNAMIC SETTING

In this section, we introduce the basic concepts and fix our notation. We mainly borrow it from our previous work [19].

A dynamic instance of a query Q is a pair (D, α), where D is a database over a finite domain and α is a sequence of updates to D, i.e. a sequence of insertions and deletions of tuples over D. The dynamic query Dyn(Q) yields as result the relation that is obtained by first applying the updates from α to D and then evaluating the query Q on the resulting database.

The database resulting from applying an update δ to a database D is denoted by δ(D). The result α(D) of applying a sequence of updates α = δ1...δm to a database D is defined by α(D) ≡ δm(...(δ1(D))...).

Dynamic programs, to be defined next, consist of an initialization mechanism and an update program. The former yields, for every (input) database D, an initial state with initial auxiliary data. The latter defines the new state of the dynamic program for each possible update δ.

A dynamic schema is a tuple (τin, τaux) where τin and τaux are the schemas of the input database and the auxiliary

---

In [19] a dynamic schema had an additional schema for an extra database with built-in relations. As in this paper, we do not restrict auxiliary relations in any way and allow arbitrary initialization, we do not need built-in relations.
database, respectively. In this work all schemata are purely relational, although all results also hold for input schemata with constants. We always let $\tau \equiv \tau_{in} \cup \tau_{aux}$.

**Definition 1.** (Update program) An update program $P$ over dynamic schema $(\tau_{in}, \tau_{aux})$ is a set of first-order formulas (called update formulas in the following) that contains, for every $R \in \tau_{aux}$ and every $\delta \in \{\text{INS}, \text{DEL}, \text{S}\}$ with $S \in \tau_{in}$, an update formula $\phi^\delta_R(\bar{x}; \bar{y})$ over the schema $\tau$ where $\bar{x}$ and $\bar{y}$ have the same arity as $S$ and $R$, respectively.

A program state $S$ over dynamic schema $(\tau_{in}, \tau_{aux})$ is a structure $(D, I, A)$ where $D$ is a finite domain, $I$ is a database over the input schema (the current database) and $A$ is a database over the auxiliary schema (the auxiliary database).

The semantics of update programs is as follows. For an update $\delta(\bar{a})$, where $\bar{a}$ is a tuple over $D$, and program state $S = (D, I, A)$ we denote by $P_S(I)$ the state $(D, \delta(D), A')$, where $A'$ consists of relations $R' \equiv \{\bar{b} \mid S \models \phi^\delta_R(\bar{a}; \bar{b})\}$. The effect $P_S$ of an update sequence $\alpha = \delta_1 \ldots \delta_m$ to a state $S$ is the state $P_{S_{\alpha}}(\ldots(P_{\delta_1}(S))\ldots)$.

**Definition 2.** (Dynamic program) A dynamic program is a triple $(P, \text{Init}, Q)$, where

- $P$ is an update program over some dynamic schema $(\tau_{in}, \tau_{aux})$,
- $\text{Init}$ is a mapping that maps $\tau_{in}$-databases to $\tau_{aux}$-databases, and
- $Q \in \tau_{aux}$ is a designated query symbol.

A dynamic program $P = (P, \text{Init}, Q)$ maintains a dynamic query $\text{Dyn}(Q)$ if, for every dynamic instance $(D, \alpha)$, the relation $Q(\alpha(D))$ coincides with the query relation $Q^S$ in the state $S = P_S(\text{Init}(D))$, where $S_{\text{Init}}(D)$ is the initial state, i.e., $S_{\text{Init}}(D) \equiv (D, \text{Init}(D))$.

Several dynamic settings and restrictions of dynamic programs have been studied in the literature (see e.g. [16, 8, 10, 9]). Possible parameters are, for instance

- the logic in which update formulas are expressible,
- whether, in dynamic instances $(D, \alpha)$, the initial database $D$ is always empty; and
- whether the initialization mapping $\text{Init}_{aux}$ is permutation-invariant (short: invariant), that is, whether $\pi(\text{Init}_{aux}(D)) = \text{Init}_{aux}(\pi(D))$ holds, for every database $D$ and permutation $\pi$ of the domain.

We refer to the introduction for a discussion of the choices made in the following definition.

**Definition 3.**(DynC) For a class $C$ of formulas, let DynC be the class of all dynamic queries that can be maintained by dynamic programs with formulas from $C$ and arbitrary initialization mapping.

In particular DynFO is the class of all dynamic queries that can be maintained by first-order update formulas. DynProp is the subclass of DynFO, where update formulas are not allowed to use quantifiers.

We note that arbitrary, (possibly) non-uniform initialization mappings permit to maintain undecided queries, even when the logic for expressing update formulas is very weak. Allowing arbitrary initialization mappings in Definition 5 helps us to concentrate on the *maintenance* aspect of dynamic complexity and it helps keeping proofs short. All our results also hold for FO-definable initialization mappings on ordered domains.

### 3. FRAGMENTS OF DynFO

In this section we study the relationship between the variants of dynamic conjunctive queries. This continues the study of fragments of first-order logic that has been started in [19].

We first give formal definitions of the classes of queries we are interested in:

- **CQ**: the class of conjunctive queries, that is, queries expressible by first-order formulas of the form $\varphi(\bar{x}) = \exists \bar{y} \psi$, where $\psi$ is a conjunction of atomic formulas.
- **UCQ**: the class of all unions of conjunctive queries, that is, queries expressible by formulas of the form $\bigvee \exists \bar{y} \psi$, where $\psi$ is a conjunction of atomic formulas.

We note that safety of queries is not an issue in this paper: we use queries as update formulas only and we can always assume that, for each required arity, there is an auxiliary “universal” relation containing all tuples of this arity over the active domain which could be used to make queries syntactically safe.

The classes CQ and UCQ can be extended by additionally allowing negated atoms, resulting in CQ$^-$ and UCQ$^-$, or they can be restricted by disallowing quantification. It is well known that UCQ$^-$ and 3$\exists$FO, the class of queries expressible by existential first-order formulas, coincide, but otherwise, all these classes are distinct. Furthermore, other quantification patterns than 3$\exists$ can be considered, like $\forall^n$ or arbitrary quantification.

The main goal of this section is to show that the relationship of these classes in the dynamic setting is much simpler than in the static setting. We prove that dynamic classes collapse as indicated in the left part of Figure 1. More precisely, we show the following two theorems regarding the second and the third fragment in the left part of Figure 1.

**Theorem 3.1.** Let $Q$ be a query. Then the following statements are equivalent:

(a) $Q$ can be maintained in DynUCQ$^-$.
(b) $Q$ can be maintained in DynCQ$^-$.
(c) $Q$ can be maintained in Dyn3$\exists$FO.
(d) $Q$ can be maintained in Dyn$\forall^n$FO.

**Theorem 3.2.** Let $Q$ be a query. Then the following statements are equivalent:

(a) $Q$ can be maintained in DynUCQ.
(b) $Q$ can be maintained in DynCQ.

Using the same technique as is used for removing unions from dynamic unions of conjunctive queries, a normal form for DynFO can be obtained. The class DynFO$^+$ contains all queries maintainable by a program whose update formulas are in prenex normal form where the quantifier-free part is a conjunction of atoms. The following theorem improves Theorem 15 from [19].

**Theorem 3.3.** Let $Q$ be a query. Then the following statements are equivalent:

(a) $Q$ can be maintained in DynFO.
(b) \( Q \) can be maintained in DynFO\(^*\).

Before we turn to the proofs of these theorems, we discuss the proof techniques that will be used.

For showing that a class DynC of queries is contained in a class DynNC\(^*\), it is sufficient to construct, for every dynamic program with update queries from class C, an equivalent dynamic program with update queries from class C\(^*\). In cases where C \( \subseteq \) C\(^*\) this can also be seen as constructing a C\(^*\)-normal form for C-programs.

Most of the proofs for the collapse of two dynamic classes in this paper are not very deep. Indeed, most of them use one or more of the following (easy) techniques.

The replacement technique ([19]) is used to remove subformulas of a certain kind from update formulas and to replace their "meaning" by additional auxiliary relations. In this way, we often can remove negations (choose negative literals as subformulas, see the proof of Theorem 3.5) and disjunctions (see proof of Lemma 3.7) from update formulas.

The preprocessing technique is used to convert (more) complicated update formulas into easier update formulas by splitting the computation performed by the complicated update formula into two parts; one of them performed by the initialization mapping and stored in an additional auxiliary relation, the other one performed by the easier update formula using the pre-computed auxiliary relation. Applications of this technique are the removal of unions from dynamic unions of conjunctive queries (see example below) as well as proving the equivalence of semantics for dynamic conjunctive queries with negations (see Lemma 4.6).

Example 1. We consider the update formula

\[ \phi^{\text{R}}(u; x) = \exists y(U(x, y) \lor V(x, u)) \]

for a unary relational symbol \( R \). We aim at an equivalent update formula \( \phi'^{\text{R}}(u; x) \) without disjunction. The idea is to store a "disjunction blueprint" in a precomputed auxiliary relation. Applications of this technique are the removal of unions from dynamic unions of conjunctive queries (see example below) as well as proving the equivalence of semantics for dynamic conjunctive queries with negations (see Lemma 4.6).

The squirrel technique maintains additional auxiliary relations that reflect the state of some auxiliary relation after every possible single update (or short update sequence).\(^3\)

For example, if a dynamic program contains a relation symbol \( R \) then a fresh relation symbol \( R_{\text{ins}} \) can be used, such that the interpretation of \( R_{\text{ins}} \) contains the content of \( R \) after update ins (for every possible insertion tuple). Of course, \( R_{\text{ins}} \) has higher arity than \( R \), as it takes the actual inserted tuple into account. Sample applications of this technique are the removal of quantifiers from some update formulas (see the following example and Lemma 3.4) and the maintenance of first-order queries in DynCQ\(^-\) (see Theorem 5.1).

Example 2. Consider the update formula

\[ \phi_{\text{ins}}^{Q}(u_1; x) = \exists y(Q(x) \lor \neg S(u_1, y)) \]

for the query symbol \( Q \) of some dynamic program \( P \). In order to obtain a quantifier-free update formula for \( Q \) after insertion of an arbitrary tuple we maintain the relation \( Q_{\text{ins}}(\cdot, \cdot) \) that contains a tuple \((a, b)\) if and only if \( b \) would be in \( Q \) in the next state, after insertion of \( a \). Similarly for \( S \) and deletions.

Then the update formula \( \phi_{\text{ins}}^{Q} \) can be replaced by the quantifier-free formula \( \phi_{\text{ins}}^{Q}(u_1; x) \equiv Q_{\text{ins}}(u_1, x) \). The relation \( Q_{\text{ins}} \) can be updated via

\[ \phi_{\text{ins}}^{Q}(u_1; u_1, x) \equiv \exists y(Q_{\text{ins}}(u_0, x) \lor \neg S_{\text{ins}}(u_0, u_1, y)) \]

\[ \phi_{\text{del}}^{Q}(u_1; u_1, x) \equiv \exists y(Q_{\text{del}}(u_0, x) \lor \neg S_{\text{del}}(u_0, u_1, y)) \]

and similarly, for the other new auxiliary relations.

We note that, in this example, the application of the technique does not eliminate all quantifiers in the program (in fact, it removes one and introduces two new formulas with quantifiers), but it removes quantification from the update formula for a particular relation. Removing quantification from the update formulas of the query relation will turn out to be useful in the proofs of Lemmata 3.7, 3.9 and 4.6.

The proof of the following technical lemma is based on this example. For an arbitrary quantifier prefix \( \mathcal{Q} \subseteq \{\exists, \forall\}^* \) let \( \mathcal{Q}\text{FO} \) be the class of queries expressible by formulas with quantifier prefix \( \mathcal{Q} \). If \( \mathcal{Q} \) is a substring of \( \mathcal{Q'} \) and a query \( \mathcal{Q} \) is in \( \mathcal{Q}\text{FO} \) then trivially \( \mathcal{Q} \) is in \( \mathcal{Q}'\text{FO} \) as well.

Lemma 3.4. Let \( \mathcal{Q} \) be an arbitrary quantifier prefix. For every DynQFO-program there is an equivalent DynQFO-program \( \mathcal{P} \) such that the update formulas for the designated query symbol of \( \mathcal{P} \) consist of a single atom.

Before we turn to dynamic conjunctive queries, we recall some results from [19]. We denote by PropCQ, PropUCQ, PropCQ\(^-\) and PropUCQ\(^-\) the classes of queries definable by conjunctive queries (and their extensions) but without quantification.

The first two results from [19] that we recall, are negation-free normal forms for DynFO and for DynProp, the class of queries maintainable by quantifier-free first-order formulas.

Theorem 3.5 ([19]). (a) Every DynFO-program has an equivalent negation-free DynFO-program.

(b) Every DynProp-program has an equivalent DynPropUCQ-program.

\( ^3 \)Squirrels usually make provisions for every possible future.
Proof idea. This theorem is a generalization of Theorem 6.6 from [14]. Given a dynamic program \( P \), the simple idea is to maintain, for every auxiliary relation \( R \) of \( P \), an additional auxiliary relation \( \hat{R} \) for the complement of \( R \).  

Furthermore, there is a disjunction-free normal form for DynProp.

Theorem 3.6 ([19]). Every DynProp-program has an equivalent DynPropCQ\(^-\)-program.

Proof idea. The update formulas of a given DynProp-program \( P \) can be assumed to be in conjunctive normal form. An equivalent DynProp\(\neg\)-program is obtained by maintaining an additional relation \( R_{\neg C} \), for every (disjunctive) clause \( C \) occurring in any update formula of \( P \).

With these additional relations, every update formula
\[
\phi = C_1(\vec{x}_1) \land \ldots \land C_k(\vec{x}_k)
\]
with clauses \( C_1(\vec{x}_1), \ldots, C_k(\vec{x}_k) \) can be replaced by the conjunctive formula
\[
\neg R_{\neg C_1}(\vec{x}_1) \land \ldots \land \neg R_{\neg C_k}(\vec{x}_k)
\]
The new auxiliary relations \( R_{\neg C} \) themselves can be maintained by viewing \( \neg C \) as a conjunction of atoms and taking the conjunction of all the conjunctive update formulas for the literals\(^a\) in \( \neg C \).

The two normal forms for DynProp can be seen as collapse results for quantifier-free conjunctive queries. Together they state that the dynamic classes DynPropUCQ, DynPropCQ\(^-\) and DynPropUCQ\(^-\) coincide with DynProp.

Now we present new collapse results for dynamic conjunctive queries. First, we prove that in the dynamic setting disjunctions can be simulated by existential quantifiers, that is DynUCQ and DynCQ\(^-\) as well as DynUCQ\(^-\) and DynCQ\(^-\) coincide. However, we can not apply the idea of the proof of Theorem 3.6 directly for this result, since UCQ and UCQ\(^-\) are not closed under negations.

Lemma 3.7. (a) For every DynUCQ\(^-\)-program there is an equivalent DynUCQ\(^-\)-program.
(b) For every DynUCQ-program there is an equivalent DynCQ\(^-\)-program.
(c) For every DynFO-program there is an equivalent DynFO\(^-\)-program.

Proof. We first prove the statements for domains with at least two elements and show how to drop this restriction afterwards. The construction uses the idea from Example 1. We give it for (a) but, as it does not introduce any negation operators it works for (b) as well. For (c) one starts from a negation-free DynFO\(^-\)-program and uses the same construction used for (a).\(^7\)

Let \( \mathcal{P} = (P, \text{Init}, Q) \) be a DynUCQ\(^-\)-program over schema \( \tau \). Without loss of generality, we assume that the quantifier-free parts of all update formulas of \( \mathcal{P} \) are in disjunctive normal form. We convert \( \mathcal{P} \) into an equivalent DynUCQ\(^-\)-program \( \mathcal{P}' \) whose update formulas are in prefix normalform with the form \( \bigwedge L_i(\vec{x}) \land T(\vec{y}) \), where \( L_i \) are arbitrary literals over the modified schema \( \hat{\tau} \) and the symbols \( T \) are fresh auxiliary relation symbols.

The program \( \mathcal{P}' = (P', \text{Init}', Q) \) is over schema \( \tau' = \tau \cup \hat{\tau} \cup \tau_\delta \), where \( \hat{\tau} \) contains a \((k+1)\)-ary relation symbol \( \hat{R} \) for every \( k \)-ary relation symbol \( R \in \tau \); and \( \tau_\delta \) contains a relation symbol \( T_{S,\delta} \) for every relation symbol \( S \in \tau \cup \hat{\tau} \) and every update operation \( \delta \).

The intention for relation symbols from \( \tau' \) is as follows. The relation symbols \( R \in \tau \) shall always be interpreted as in \( \mathcal{P} \). The intention of \( \hat{R} \in \hat{\tau} \) is, on one hand, to contain a “copy of \( R' \)” (in tuples with first component \( c \), for some fixed element \( c \)) and on the other hand, to have a guarantee that \( \hat{R} \) is non-empty. The latter is strongly ensured by enforcing all tuples that do not have \( c \) as first component to be in \( \hat{R} \), and by \(|D| \geq 2\):\(^8\)

\[
\hat{R}^S \overset{\text{def}}{=} \{ (c, \vec{a}) \mid \vec{a} \in R^S \} \cup \{ (d, \vec{a}) \mid d \neq c \text{ and } \vec{a} \in D^k \}
\]

The relations \( T_{S,\delta} \) will be used as in Example 1.

Now we construct the update formulas for program \( \mathcal{P}' \). Let \( R \in \tau \) and \( \delta \) be an update. Further let
\[
\phi^R_{\delta}(\vec{u}; \vec{x}, \vec{y}) = \exists y(C_{\delta}(\vec{u}, \vec{x}, \vec{y})) \ldots \lor C_{\delta}(\vec{u}, \vec{x}, \vec{y})
\]
be the update formula of \( R \) with respect to \( \delta \) in \( \mathcal{P} \), where \( C_{\delta} \) is a conjunction of literals. For
\[
C_i(\vec{u}, \vec{x}, \vec{y}) = L_1(\vec{v}_1) \land \ldots \land L_m(\vec{v}_m)
\]
we define
\[
\hat{C}_i(\vec{z}, \vec{z}_1) \overset{\text{def}}{=} L_1(\vec{v}_1) \land \ldots \land L_m(\vec{v}_m)
\]
where \( L_j = \hat{R} \) if \( L_j = R \) and \( L_j = -\hat{R} \) if \( L_j = -R \).

The update formula \( \psi^R_{\delta}(\vec{u}; \vec{x}, \vec{y}) \) for \( R \in \hat{\tau} \) in \( \mathcal{P}' \) is
\[
\psi^\hat{R}_{\delta}(\vec{u}; \vec{x}, \vec{y}) \overset{\text{def}}{=} \exists y_1 \exists z_1 \exists z_k \exists z_k \exists z_k
\]
\[
\hat{C}_i(\vec{z}_1, \vec{z}_1) \land \cdots \land \hat{C}_k(\vec{z}_k, \vec{z}_k)
\]
\[
\land T_{S,\delta}(\vec{y}, \vec{z}_1, \vec{z}_1, \ldots, \vec{z}_k, \vec{z}_k, \vec{u}, \vec{x}, \vec{y})
\]
\[
\land T_{R,\delta}(\vec{y}, \vec{z}_1, \vec{z}_1, \ldots, \vec{z}_k, \vec{z}_k, \vec{u}, \vec{x}, \vec{y})
\]
To ensure equivalence of this program with the original program, the relations \( T_{S,\delta} \) are defined as follows.

- \( T_{S,\delta} \) contains all tuples\(^8\) \((\vec{y}, \vec{z}_1, \vec{z}_1, \ldots, \vec{z}_k, \vec{z}_k, \vec{u}, \vec{x}, \vec{y})\), for which, for some \( i \), \( \vec{z}_i = c \) and \( \vec{z}_i = (\vec{u}, \vec{x}, \vec{y}) \).
- \( T_{R,\delta} \) contains all tuples \((\vec{y}, \vec{z}_1, \vec{z}_1, \ldots, \vec{z}_k, \vec{z}_k, \vec{u}, \vec{x}, \vec{y})\), for which
  \[- \vec{x} \neq c, \text{ or}\]

\(^a\)For this step it is needed that \( P \) contains, for every auxiliary relation \( R \), an additional auxiliary relation \( \hat{R} \) for the complement of \( R \). This can be ensured by the technique from Theorem 3.5.

\(^7\)More precisely, replace the quantifier prefix \( \exists y \) used throughout the construction of (a) by a general quantifier-prefix \( \exists y_1 \forall y_2 \ldots \).

\(^8\)For simplicity, we reuse variable names as element names.
These are initialized as intended by simple quantifier-free formulas (but with disjunction). Their interpretation is never changed, that is, for every $T_{\delta\beta}$, both update formulas reproduce the current value of $T_{\delta\beta}$.

The initialization for relation symbols from $\tau$ and $\vec{\tau}$ is straightforward. Auxiliary relation symbols $R \in \tau$ are initialized as in $\mathcal{P}$; and auxiliary relation symbols $\vec{R} \in \vec{\tau}$ are initialized by $\text{Init}'$ analogously to $\text{Init}$ but respecting Equation (1).

This concludes the proof of (a), (b) and (c) for domains with at least two elements. The restriction on the size of the domains can be dropped as follows. In all three cases the idea is to make a case distinction on the size of the domain in the update formulas of the designated query symbol.

To this end, we first construct a DynPropCQ-program $\mathcal{P}'' = (\mathcal{P}', \text{Init}', Q''')$ over schema $\vec{\tau}'' = \tau'' \cap \vec{\tau}'' = \emptyset$ which is equivalent to $\mathcal{P}$ over databases with domains of size one. Then we construct a program $\mathcal{P}'''$ equivalent to $\mathcal{P}$ by combining $\mathcal{P}''$ and the program $\mathcal{P}'''$ for domains of size at least two.

For the construction of $\mathcal{P}'''$ we observe that every relation of a database over a single element domain $D = \{a\}$ contains either exactly one tuple, namely $(a, \ldots, a)$, or no tuple at all. Thus every such relation $R$ corresponds to a 0-ary relation $R_0$ where $R_0$ is true if and only if $(a, \ldots, a) \in R$. Hence, by Lemma 3.8, there is a DynPropCQ-program equivalent to $\mathcal{P}$ for databases with domains of size one.

To combine $\mathcal{P}''$ and $\mathcal{P}'''$ we use two different approaches, one for (a) and one for (b) and (c).

First we consider (a). To this end, we can assume, by Lemma 3.4, that the update formulas for the query relations $Q'$ and $Q''$ of $\mathcal{P}'$ and $\mathcal{P}''$ consist of single atoms, respectively. We construct an intermediate program $\tilde{\mathcal{P}} = (\tilde{P}, \text{Init}, \tilde{Q})$ over schema $\vec{\tau} = (\tilde{Q}, \vec{\tau}''')$ where $\vec{\tau}$ is a fresh 0-ary relation symbol. The intention is that interpretations of symbols in $\vec{\tau}$ and $\vec{\tau}''$ are as in $\mathcal{P}'$ and $\mathcal{P}''$, respectively, and that $U$ is interpreted by true if and only if the domain is of size one. The initializations are accordingly.

Thus all update formulas of $\tilde{\mathcal{P}}$ for relation symbols from $\vec{\tau}$ and $\vec{\tau}''$ are as in $\mathcal{P}'$ and $\mathcal{P}'''$ (and thus disjunction-free).

The program $\mathcal{P}'''$ is obtained from $\tilde{\mathcal{P}}$ by removing disjunctions from $\phi^Q_3$ using the method 10 from the proof of Theorem 3.6. For example, the first clause is replaced by $\neg R(Q \cup \neg Q''')$ where $R(Q \cup \neg Q''')$ is a fresh auxiliary relation symbol intended to be always interpreted by the result of the query $\neg (\phi^{Q'}_3 \lor \phi^{Q''}_3)$. The update formula for $R_{\neg (Q \cup \neg Q''')}$ after an update $\delta$ is $\neg \phi^{Q'}_3 \land \neg \phi^{Q''}_3$. It is disjunction-free since, under our assumption, $\phi^{Q'}_3$ and $\phi^{Q''}_3$ both consist of a single atom. This concludes the proof of (a).

The program $\mathcal{P}'''$ for (b) and (c) is over schema $\vec{\tau}''' = \vec{\tau}'' \cup \vec{\tau}''$. Again all update formulas of $\mathcal{P}'''$ for relation symbols from $\vec{\tau}$ and $\vec{\tau}'''$ are as in $\mathcal{P}'$ and $\mathcal{P}'''$ and

This method cannot be used for DynCQ and DynFO$^\wedge$.

The case distinction is delegated to the initialization mapping. Recall that the size of the domain is fixed when the auxiliary relations are initialized. The initialization mapping $\text{Init}'''$ is as follows. If $|D| = 1$

\[
\text{Init}'''(R) = \begin{cases} 
\text{Init}'(Q') & \text{for } R = Q'', \\
D^k & \text{for } R \in \tau', \\
\text{Init}'''(R'') & \text{for } R \in \tau''
\end{cases}
\]

If $|D| \geq 2$ then

\[
\text{Init}'''(R) = \begin{cases} 
\text{Init}'(Q') & \text{for } R = Q'', \\
\text{Init}'''(R'') & \text{for } R \in \tau', \\
D^k & \text{for } R \in \tau''
\end{cases}
\]

Thus $\text{Init}'''$ selects either $\phi^{Q'}_3$ or $\phi^{Q''}_3$, depending on the size of the domain. If $|D| = 1$ then $\phi^{Q'}_3$ always evaluates to true whereas $\phi^{Q''}_3$ yields the same value as in $\mathcal{P}'''$, and vice versa for $|D| \geq 2$. As update formulas do not use negation, all relations in the program, that is initialized to “true” ($\mathcal{P}'$ or $\mathcal{P}'''$) remain “full” throughout. This concludes the proof of (b).

0-ary relations can either be true (containing the empty tuple) or false (not containing the empty tuple and thus being empty); thus 0-ary atoms are basically propositional variables. Queries on 0-ary databases are therefore basically families of Boolean functions, one for each domain size. Such queries are not very interesting from the perspective of databases, but we need to show the following lemma as we used it in the previous proof. As quantification in queries on 0-ary databases is useless, every FO query can be expressed by a quantifier-free formula and therefore can be maintained in DynProp. Yet, even more general, all queries on a 0-ary database can be maintained by even more restricted dynamic programs. Let DynPropCQ be the class of dynamic queries definable by update programs whose update formulas are solely conjunctions of atoms (i.e. no negations, no disjunctions and no quantifiers are allowed).

**Lemma 3.8.** Every query on a 0-ary database can be maintained by a DynPropCQ-program.

**Proof.** Let $\tau_m$ be an input schema with 0-ary relation symbols $A_1, \ldots, A_k$. Further let $Q_1, \ldots, Q_m$ be an enumeration of all $m = 2^k$ many queries on $\tau_m$. We actually show that all of them can be maintained by one DynPropCQ-program $\mathcal{P}$ with auxiliary schema $\tau_{aux} = \{R_1, \ldots, R_m\}$ maintaining $Q_i$ in $R_i$, for every $i \in \{1, \ldots, m\}$.

To this end, let $\phi_1, \ldots, \phi_m$ be propositional formulas over $\tau_m$ such that $\phi_i$ expresses $Q_i$, and each $\phi_i$ is in conjunctive normal form. Without loss of generality, no clause contains $A_i$ and $\neg A_i$ for any $A_i \in \tau_m$ and any $\phi_i$. As $\tau_{aux}$ contains a relation symbol, for every propositional formula over $A_1, \ldots, A_k$, it contains, in particular, an auxiliary relation symbol $RC_i$ for every disjunctive clause over $A_1, \ldots, A_k$.

The update formulas for $R_i$ after changing input relation $A_i$ can be constructed as follows. Let $\mathcal{C}$ be the set of clauses of $\phi_i$, i.e. $\phi_i = \bigwedge_{C \subseteq C} C$. We denote by $C_{A_i}, C_{\neg A_i}$ and $C_{A_i}$ the subsets of $\mathcal{C}$ whose clauses contain $A_i$, $\neg A_i$ and neither $A_i$ nor $\neg A_i$, respectively.

10This cannot be guaranteed for DynUCQ$^\neg$.
If $A_l$ becomes true by an update then $\varphi_j$ evaluates to true if all clauses in $C_{A_l}$ and all clauses $C \setminus \{\neg A_l\}$ with $C \in C_{A_l}$ evaluated to true before the update (clauses in $C_{A_l}$ will evaluate to true after enabling $A_l$).

If $A_l$ becomes false by an update then $\varphi_j$ evaluates to true if all clauses in $C_{A_l}$ and all clauses $C \setminus \{\neg A_l\}$ with $C \in C_{A_l}$ evaluated to true before the update (clauses in $C_{A_l}$ will evaluate to true after disabling $A_l$).

Therefore the update formulas for $R_j$ after updating $A_l$ can be defined as follows:

$\phi_{\text{del},A_l}^{R_j} \defeq \bigwedge_{C \in C_{A_l}} R_C \land \bigwedge_{C \in C_{A_l}^+} R_{C \setminus \{\neg A_l\}}$

$\phi_{\text{ins},A_l}^{R_j} \defeq \bigwedge_{C \in C_{A_l}} R_C \land \bigwedge_{C \in C_{A_l}^-} R_{C \setminus \{A_l\}}$

The initialization is straightforward. The correctness of this construction can be proved by induction on the length of update sequences.

Finally we prove that $\text{Dyn}^\exists^*-\text{FO} = \text{Dyn}^\forall^*-\text{FO}$, and therefore that unions of conjunctive queries with negation coincide with $\text{Dyn}^\forall^*-\text{FO}$ in the dynamic setting. The proof uses the replacement technique to maintain the complements of the auxiliary relations used in the $\text{Dyn}^\exists^*-\text{FO}$-program via $\text{Dyn}^\forall^*-\text{FO}$-formulas. A small complication arises from the fact, that the query relation (and not its complement) has to be maintained. This is solved by ensuring that the update formulas of the query relation are atomic.

A slightly more general result can be shown.

**Lemma 3.9.** Let $\mathcal{Q}$ be an arbitrary quantifier prefix. A query can be maintained in $\text{Dyn}^\forall^*\text{FO}$ if and only if it can be maintained in $\text{Dyn}^\forall^*\text{FO}$.

Now Theorems 3.1 and 3.2 follow immediately from Lemma 3.7 and Lemma 3.9.

### 4. $\Delta$-Semantics

So far we considered a semantics where the new version of the auxiliary relations is redefined, after each update, from scratch by formulas that are evaluated on the structure with the current auxiliary relations. We refer to this as *absolute semantics* in the following.

However, in the context of view maintenance, one usually expects only few auxiliary tuples to change after an update. Therefore it is common to express the new version of the auxiliary relations in terms of the current relations and some “Delta”, that is, a (small) relation $R^+$ of tuples to be inserted into $R$ and a (small) relation $R^-$ of tuples to be removed from $R$ (with $R^+ \cap R^- = \emptyset$). The updated auxiliary relation $R'$ is then defined by

$R' \defeq (R \cup R^+) \setminus R^-$

We refer to this semantics as *$\Delta$-semantics*. This is the semantics usually considered in view maintenance. As already stated in the introduction, absolute and $\Delta$-semantics can only be different if the underlying update language is not closed under Boolean operations.

Next we formalize $\Delta$-semantics via $\Delta$-update programs which provide formulas defining the relations $R^+$ and $R^-$, for every auxiliary relation $R$.

**Definition 4.** (\(\Delta\)-Update program) A $\Delta$-update program $\mathcal{P}$ over dynamic schema $(\tau_n, \tau_{\text{aux}})$ is a set of first-order formulas (called $\Delta$-update formulas in the following) that contains, for every $R \in \tau_{\text{aux}}$ and every $\delta \in \{\text{ins}, \text{del}\}$ with $S \in \tau_n$, two formulas $\phi^R_{\text{ins}}(\vec{u}; \vec{x})$ and $\phi^R_{\text{del}}(\vec{u}; \vec{x})$ over the schema $\tau$ where $\vec{u}$ and $S$ have the same arity, $\vec{x}$ and $R$ have the same arity, and $\phi^R_{\text{ins}} \land \phi^R_{\text{del}}$ is unsatisfiable.

The semantics of $\Delta$-update programs is as follows. For an update $\delta = \delta(\vec{a})$ and program state $S = (D, \delta(D), \mathcal{A})$, where the relations $R'$ of $\mathcal{A}'$ are defined by

$R' = (R \cup \{\vec{b} \mid S \models \phi^R_{\text{ins}}(\vec{a}; \vec{b})\}) \setminus \{\vec{b} \mid S \models \phi^R_{\text{del}}(\vec{a}; \vec{b})\}$

The effect of an update sequence on a state, dynamic $\Delta$-programs and so on are defined like their counterparts in absolute semantics except that $\Delta$-update programs are used instead of update programs.

**Definition 5.** (\(\Delta\)-DynC) For a class $\mathcal{C}$ of formulas, let $\Delta$-DynC be the class of all dynamic queries that can be maintained by dynamic $\Delta$-programs with formulas from $\mathcal{C}$ and arbitrary initialization mapping.

We note that the definitions above do not require that $R^+ \cap R^- = \emptyset$ or $R^+ \subseteq R$. That is, $R^+$ might contain tuples that are already in $R$, and $R^-$ might contain tuples that are not in $R$. However, in all proofs below, we construct only $\Delta$-update formulas that guarantee these additional properties. As a consequence, for the considered fragments, the expressive power is independent of this difference.

The goal of this section is to prove the remaining results of Figure 1, that is, the collapse results depicted in the right part of the figure and the correspondences between absolute semantics and $\Delta$-semantics.

**Theorem 4.1.** Let $\mathcal{Q}$ be a query. Then the following statements are equivalent:

(a) $\mathcal{Q}$ can be maintained in $\Delta$-DynUCQ$^\forall$.

(b) $\mathcal{Q}$ can be maintained in $\Delta$-DynUCQ.

(c) $\mathcal{Q}$ can be maintained in $\Delta$-DynCQ$^\forall$.

(d) $\mathcal{Q}$ can be maintained in $\Delta$-DynCQ.

(e) $\mathcal{Q}$ can be maintained in $\Delta$-DynCQ$^\exists$.

(f) $\mathcal{Q}$ can be maintained in $\Delta$-Dyn$^\forall\forall$-FO.

**Theorem 4.2.** Let $\mathcal{Q}$ be a query. Then the following statements are equivalent:

(a) $\mathcal{Q}$ can be maintained in $\text{Dyn}^\forall^\mathcal{C}$.

(b) $\mathcal{Q}$ can be maintained in $\Delta$-DynUCQ$^\forall$.

The technique used for removing unions from dynamic unions of conjunctive queries under $\Delta$-semantics can be used to obtain a $\Delta$-DynFO$^\wedge$ normal form for $\Delta$-DynFO-programs.

**Theorem 4.3.** Let $\mathcal{Q}$ be a query. Then the following statements are equivalent:

(a) $\mathcal{Q}$ can be maintained in $\Delta$-DynFO.
(b) \( \mathcal{Q} \) can be maintained in \( \Delta\text{-DynFO}^\land \).

We state some basic facts about dynamic programs with \( \Delta \)-semantics before proving those theorems. The following lemma establishes the obvious fact that the absolute semantics and \( \Delta \)-semantics coincide in expressive power for dynamic classes closed under boolean operations. We observe that the proof does not work for (extensions of) conjunctive queries. Later we will see how to extend the result to conjunctive queries.

Lemma 4.4. Let \( \mathcal{C} \) be some fragment of first-order logic closed under the boolean operations \( \{ \lor, \land, \neg \} \). Then for every query \( \mathcal{Q} \) the following are equivalent:

(a) There is a \( \text{DynC} \)-program that maintains \( \mathcal{Q} \).

(b) There is a \( \Delta\text{-DynC} \)-program that maintains \( \mathcal{Q} \).

Proof. From an \( \text{DynC} \)-update formula \( \phi_0^R \), the \( \Delta\text{-DynC} \)-update formulas are defined as

\[
\phi_\delta^{R^+} (\bar{u}; \bar{x}) \overset{\text{def}}{=} \phi_\delta^R (\bar{u}; \bar{x}) \land \neg R(\bar{x}) \\
\phi_\delta^{R^-} (\bar{u}; \bar{x}) \overset{\text{def}}{=} \neg \phi_\delta^R (\bar{u}; \bar{x}) \land R(\bar{x})
\]

From a \( \Delta\text{-DynC} \)-update formula \( \phi_\delta^{R^+} \) and \( \phi_\delta^{R^-} \), an \( \Delta\text{-DynC} \)-update formula is obtained via

\[
\phi_\delta^R (\bar{u}; \bar{x}) \overset{\text{def}}{=} (R(\bar{x}) \lor \phi_\delta^{R^+} (\bar{u}; \bar{x})) \land \neg \phi_\delta^{R^-} (\bar{u}; \bar{x})
\]

Removing negations in dynamic programs with \( \Delta \)-semantics is straightforward using the replacement technique, since the complement \( \bar{R} \) of an auxiliary relation \( R \) can be maintained by exchanging the formulas \( \phi_\delta^{R^+} \) and \( \phi_\delta^{R^-} \). Observe that in contrast to absolute semantics this works for arbitrary query classes, even if they are not closed under complementation, and in particular for (extensions of) conjunctive queries.

Lemma 4.5. Let \( \mathcal{C} \) be some fragment of first-order logic. If a query \( \mathcal{Q} \) can be maintained in \( \Delta\text{-DynC} \) then \( \mathcal{Q} \) can be maintained in negation-free \( \Delta\text{-DynC} \).

Proof. The idea is again to maintain the complements for auxiliary relations. For the sake of completeness we give a full proof.

Given a dynamic \( \Delta \)-program \( \mathcal{P} \) over schema \( \tau \) we construct a dynamic \( \Delta \)-program \( \mathcal{P}' \) over schema \( \tau \cup \bar{\tau} \) where each relation symbol, for every \( \ell \)-ary relation symbol \( R \in \tau \), a fresh \( \ell \)-ary relation symbol \( \bar{R} \) is used to store the complement of \( R \).

The update formulas for \( R \in \tau \) are as in \( \mathcal{P} \). For a relation symbol \( R \in \tau \) let \( \phi_\delta^R (\bar{u}; \bar{x}) \) and \( \bar{\phi}_\delta^R (\bar{u}; \bar{x}) \) be the update formulas of \( R \). Then the update formulas for \( \bar{R} \) can be defined as follows:

\[
\phi_\delta^{\bar{R}^+} (\bar{u}; \bar{x}) = \phi_\delta^{R^-} (\bar{u}; \bar{x}) \\
\phi_\delta^{\bar{R}^-} (\bar{u}; \bar{x}) = \phi_\delta^{R^+} (\bar{u}; \bar{x})
\]

From \( \mathcal{P}' \), a negation-free dynamic \( \Delta \)-program \( \mathcal{P}'' \) can be constructed by replacing, for all \( R \in \tau \), all occurrences of \( \neg R(\bar{x}) \) in update formulas of \( \mathcal{P}' \) by \( \bar{R}(\bar{x}) \). We omit the obvious proof of correctness. \( \Box \)

We now turn towards proving the main results of this section. We first prove Theorem 4.2. Afterwards we use the connection between absolute and \( \Delta \)-semantics that it establishes as well as the adaption of Lemma 3.7 to \( \Delta \)-semantics to prove the characterization of conjunctive queries with \( \Delta \)-semantics.

The only-if-direction of Theorem 4.2 can be generalized to arbitrary quantifier prefixes. It is open whether the if-direction generalizes as well.

Lemma 4.6. Let \( \mathcal{Q} \) be an arbitrary quantifier prefix. If a query can be maintained in \( \text{DynQFO} \) then it can be maintained in \( \Delta\text{-DynQFO} \) as well.

Proof. Let \( \mathcal{P} = (\mathcal{P}, \text{Init}, \mathcal{Q}) \) be a \( \text{DynQFO} \)-program with schema \( \tau \). By Lemma 3.4 we can assume, without loss of generality, that the update formulas of \( \mathcal{Q} \) are atomic. We construct a dynamic \( \Delta\text{-DynQFO} \)-program \( \mathcal{P}' = (\mathcal{P}', \text{Init}', \mathcal{Q}') \).

The main challenge is to design update formulas of the kind \( \phi^{\mathcal{Q}'} \) without being able to complement the given update formulas because this would lead to \( \text{QFO} \)-formulas (additionally, the disjointness requirement for formulas \( \phi^{\mathcal{Q}'} \) needs to be ensured).

The basic idea is to use two copies of the auxiliary relations, both alternating between empty and useful states, such that one copy is useful for even steps and the other one for odd steps. More precisely, for every auxiliary relation \( R \) used by \( \mathcal{P} \), the program \( \mathcal{P}' \) uses two auxiliary relations \( R_{\text{even}} \) and \( R_{\text{odd}} \) with the intention that after an even sequence of updates \( R_{\text{even}} \) stores the content of \( R \) after the same sequence of updates while \( R_{\text{odd}} \) is empty. After an odd sequence of updates \( R_{\text{even}} \) is empty while \( R_{\text{odd}} \) stores the content of \( R \).

Then, for an even update, the relation \( R_{\text{even}} \) can be simply expressed as in absolute semantics (using "odd" relations) and \( R_{\text{even}} \) is empty. For an odd update \( R_{\text{even}} \) can be simply chosen as \( R_{\text{even}} \) and \( R_{\text{even}} \) is empty. Similarly for \( R_{\text{odd}} \).

In the following we give a precise construction of \( \mathcal{P}' \) over schema \( \tau_{\text{even}} \cup \tau_{\text{odd}} \cup \{ \text{Odd}(\mathcal{Q}) \} \) where \( \text{Odd} \) is a boolean relation symbol, and \( \tau_{\text{even}} \) and \( \tau_{\text{odd}} \) contain, for every k-ary relation symbol \( R \in \tau \), a k-ary relation symbol \( R_{\text{even}} \) and \( R_{\text{odd}} \), respectively. The relation \( \text{Odd} \) is used to store the parity of the number of updates performed so far.

Let \( \phi^{\mathcal{Q}'} \) be the update formula of \( R \in \tau \) for an update \( \delta \) in the dynamic program \( \mathcal{P} \). Denote by \( \phi^{\mathcal{Q}'}_{\text{even}} [\tau \rightarrow \tau_{\text{even}}] \) the formula obtained from \( \phi^{\mathcal{Q}'} \) by replacing every atom \( S(\bar{x}) \) with \( S \in \tau \) by \( S_{\text{even}}(\bar{x}) \). Analogously for \( \phi^{\mathcal{Q}'}_{\text{odd}} [\tau \rightarrow \tau_{\text{odd}}] \). Now, the update formulas for \( R_{\text{odd}} \) and \( R_{\text{even}} \) are as follows:

\[
\phi^{\text{R}_{\text{odd}}}_{\text{even}} (\bar{u}; \bar{x}) \overset{\text{def}}{=} -\text{Odd} \land \phi^{\mathcal{Q}'} (\tau \rightarrow \tau_{\text{even}})(\bar{u}; \bar{x}) \\
\phi^{\text{R}_{\text{odd}}}_{\text{odd}} (\bar{u}; \bar{x}) \overset{\text{def}}{=} \text{Odd} \land R_{\text{odd}}(\bar{x}) \\
\phi^{\text{R}_{\text{even}}}_{\text{even}} (\bar{u}; \bar{x}) \overset{\text{def}}{=} \text{Odd} \land R_{\text{even}}(\bar{x}) \\
\phi^{\text{R}_{\text{even}}}_{\text{odd}} (\bar{u}; \bar{x}) \overset{\text{def}}{=} -\text{Odd} \land R_{\text{even}}(\bar{x})
\]

Observe that all those formulas can be easily converted into \( \text{QFO} \)-formulas. The boolean auxiliary relation \( \text{Odd} \) can be updated straightforwardly.

Now, since the update formulas of \( \mathcal{Q} \) in \( \mathcal{P} \) are quantifier-free, the relation \( \mathcal{Q}' \) can be updated with the following...
quantifier-free update formulas:
\[
\phi^{\tau+}_S(\vec{u}; \vec{x}) \triangleq \phi^S_0(\vec{u}; \vec{x}) \land \\
\neg \left( (\text{Odd} \land Q_{\text{odd}}(\vec{x})) \lor (\neg\text{Odd} \land Q_{\text{even}}(\vec{x})) \right)
\]
\[
\phi^{\tau-}_S(\vec{u}; \vec{x}) \triangleq \neg \phi^S_0(\vec{u}; \vec{x}) \land \\
\left( (\text{Odd} \land Q_{\text{odd}}(\vec{x})) \lor (\neg\text{Odd} \land Q_{\text{even}}(\vec{x})) \right)
\]

The initialization mapping of \(P'\) is straightforward. Every \(R_{\text{even}} \in \tau_{\text{even}}\) is initialized with \(\text{INIT}(R)\). All \(R_{\text{odd}} \in \tau_{\text{odd}}\) are initialized with the empty relation. The relation \(\text{Odd}\) is initialized with \(\bot\), and \(Q'\) is initialized with \(\text{INIT}(Q)\). □

**Lemma 4.7.** (a) If a query can be maintained in \(\Delta\text{-DynUCQ}^\neg\) then it can be maintained in \(\text{DynUCQ}^\neg\).

(b) If a query can be maintained in \(\Delta\text{-Dyn}^\neg\text{FO}\) then it can be maintained in \(\text{Dyn}^\neg\text{FO}\) as well.

We note that the first statement could equally be expressed in terms of \(\Delta\text{-Dyn}^\neg\text{FO}\) and \(\text{Dyn}^\neg\text{FO}\).

**Proof.** We only prove (a), the proof of (b) is analogous. Let \(P = (P, \text{INIT}, Q)\) be a dynamic \(\Delta\text{-DynUCQ}^\neg\)-program over schema \(\tau\). By Lemma 4.5 we can assume, without loss of generality, that the update formulas of \(P\) are negation-free. For ease of presentation we assume that the input schema contains a single binary relation symbol \(E\).

We construct an equivalent \(\text{DynUCQ}^\neg\)-program \(P'\) using the following idea. Consider some update formulas \(\phi^R_0(\vec{u}; \vec{x})\) and \(\phi^R_0(\vec{u}; \vec{x})\) of a relation \(R \in \tau\) for an update \(\delta\) in \(P\). The naive translation into a \(\text{DynFO}\)-update formula \(\phi^R_0(\vec{u}; \vec{x})\) yields the formula

\[
\phi^R_0(\vec{u}; \vec{x}) = (R(\vec{x}) \lor \phi^R_0(\vec{u}; \vec{x})) \land \neg \phi^R_0(\vec{u}; \vec{x})
\]

which is possibly non-UCQ\(^\neg\) due to \(\neg \phi^R_0(\vec{u}; \vec{x})\). Therefore, \(P'\) maintains a relation \(R_T\) that contains all tuples \((\vec{a}, \vec{b})\) such that \(\vec{a}\) would be removed from \(R\) after applying the update \(\delta(\vec{b})\). These relations are maintained using the squirrel technique.

The dynamic program \(P'\) is over schema \(\tau \cup \tau_{\Delta}\) where \(\tau_{\Delta}\) contains a \((k + 2)\)-ary relation symbol \(R_{\Delta}\) for every \(k\)-ary relation symbol \(R \in \tau\) and every update \(\delta \in \{\text{INS}, \text{DEL}\}\) of the input relation \(E\).

The update formula for a relation symbol \(R \in \tau\) is

\[
\phi^R_0(\vec{u}; \vec{x}) \triangleq (R(\vec{x}) \lor \phi^R_0(\vec{u}; \vec{x})) \land \neg R^\neg_{\Delta}(\vec{u}, \vec{x})
\]

This formula can be translated into an existential formula in a straightforward manner.

For updating a relation \(R_{\Delta}\) after an update \(\delta_0\), the update formula \(\phi^R_{\Delta}S\) for \(R^\neg\) is used. However, since \(R^\neg_{\Delta}\) shall store tuples that have to be deleted after applying \(\delta_0\), the formula \(\phi^R_{\Delta}S\) has to be adapted to use the content of relation symbols \(S \in \tau\) after update \(\delta_0\) (instead, as usual, the content from before the update). For this purpose relation symbols \(S \in \tau\) in \(\phi^R_{\Delta}S\) need to be replaced by their update formulas as defined above.

The update formula for \(R_{\Delta}\) is

\[
\phi^R_{\Delta}\left(\vec{u}_0; \vec{u}_1, \vec{x}\right) \triangleq \phi^R_{\Delta}\left[\tau \rightarrow \phi^R\right]\left(\vec{u}_0; \vec{u}_1, \vec{x}\right)
\]

where \(\phi^R_{\Delta}\left[\tau \rightarrow \phi^R\right]\left(\vec{u}_0; \vec{u}_1, \vec{x}\right)\) is obtained from \(\phi^R_{\Delta}\) by replacing every atom \(S(\vec{x})\) by \(\phi^S_0(\vec{u}_0; \vec{x})\), as constructed above. Since by our initial assumption, \(\phi^R\) itself is an existential formula without negation and all update formulas \(\phi^S_0\) for \(S \in \tau\) are existential, the formula \(\phi^R_{\Delta}\) can be easily converted into an existential formula as well. □

**Lemma 4.8.** (a) For every \(\Delta\text{-DynUCQ}^\neg\)-program there is an equivalent \(\Delta\text{-DynCQ}^\neg\)-program.

(b) For every \(\Delta\text{-DynFO}\)-program there is an equivalent \(\Delta\text{-Dyn}^\neg\text{FO}\)-program.

Now, Theorem 4.1 follows from the Lemmata 4.5, 4.8, 4.6 and 4.7 as well as from Theorem 3.1.

5. A DYNAMIC CHARACTERIZATION OF FIRST-ORDER LOGIC

In this section we characterize first-order queries as the class of queries maintainable by non-recursive UCQ\(^\neg\)-programs and, equivalently, by non-recursive DynCQ\(^\neg\)-programs. Here \(\exists^1\text{FO}\) is the class of queries expressible by first-order formulas in prenex normal form with at most one existential quantifier and no universal quantifiers, and “non-recursive” is explained next. This characterization in combination with Theorem 3.1 yields that first-order queries can be dynamically maintained by CQ\(^\neg\)-programs.

The dependency graph of a dynamic program \(P\) with auxiliary schema \(\tau\) has vertex set \(V = \tau\) and an edge \((R, R')\) if the relation symbol \(R'\) occurs in one of the update formulas for \(R\). A dynamic program is non-recursive if it has an acyclic dependency graph (as a directed graph). For every class \(C\), non-recursive DynC refers to the set of queries that can be maintained by non-recursive DynC\(^\neg\)-programs.

The objective of this section is to prove the following theorem.

**Theorem 5.1.** For every query \(Q\) the following statements are equivalent

(a) \(Q\) can be expressed in \(\text{FO}\).

(b) \(Q\) can be maintained in non-recursive \(\text{DynFO}\).

(c) \(Q\) can be maintained in non-recursive \(\text{Dyn}^\neg\text{FO}\).

(d) \(Q\) can be maintained in non-recursive \(\text{Dyn}^\neg\text{FO}\).

With respect to the number of quantifiers in update formulas this result is optimal because the first-order definable alternating reachability query on graphs of bounded diameter cannot be maintained with quantifier-free update formulas [9]. Theorem 5.1 should be compared with the result of [9] that all \(\exists^1\text{FO}\) queries can be maintained with quantifier-free update programs extended by auxiliary functions.

Combining Theorem 5.1 with Theorem 3.1 immediately yields the following corollary.

**Corollary 5.2.** Every first-order query can be maintained in DynCQ\(^\neg\).
The rest of this section is devoted to the proof of Theorem 5.1, more precisely to the equivalence of statements (a)-(c). The equivalence with (d) follows from Theorem 3.1 and the fact that its proof does not introduce recursion when applied to a non-recursive program. It is obvious that (c) implies (b). For ease of presentation, we prove the remaining directions (a)⇒(c) and (b)⇒(a) for the input schema $\tau_m = \{E\}$ where $E$ is a binary relation symbol. The proofs can be easily adapted to general (relational) signatures.

The following example outlines the idea of the construction for the proof of (a)⇒(c).

**Example 3.** Consider the query $Q$ defined by
\[
\varphi = \exists x \forall y \left( (E(x, x) \rightarrow E(x, y)) \right) \\
\equiv \exists x \neg \exists y \neg (E(x, x) \rightarrow E(x, y))
\]
We construct a non-recursive dynamic DY\text{N}FO-program $P$ that maintains $Q$ under deletions only (for simplicity). The construction of $P$ uses the squirrel technique. It uses a separate auxiliary relation $R_\psi$ for each subformula $\psi$ obtained from $\varphi$ by strippin off a "quantifier prefix" from the existential prefix of $\varphi$. The relation $R_\psi$ reflects the possible states after a sequence of changes whose length equals the number of stripped off $\exists$-symbols.

In order to update the query relation after the deletion of an edge, we maintain an auxiliary ternary relation $R_1$ that contains the result of the query $\psi_1 \equiv \neg \exists y \neg (E(x, x) \rightarrow E(x, y))$ for every choice $a_1$ for $x$ and every (possibly deleted) edge $e_1$, that is $(a_1, e_1) \in R_1$ if and only if
\[
(V, E \setminus \{e_1\}, \{x \rightarrow a_1\}) \models \forall y (E(x, x) \rightarrow E(x, y))
\]
Then we can define $\phi_1^{Q, \psi_1}(e_1) \equiv \exists x R_1(x, e_1)$ and it only remains to find a way to update the relation $R_1$. To this end, we maintain a further relation $R_2$ that contains the result of $\psi_2 \equiv \exists y (E(x, x) \rightarrow E(x, y))$ for every choice $a_1$ for $x$ and all (possibly deleted) edges $e_1, e_2$, that is $(a_1, e_1, e_2) \in R_2$ if and only if
\[
(V, E \setminus \{e_1, e_2\}, \{x \rightarrow a_1\}) \models \exists y (E(x, x) \rightarrow E(x, y))
\]
Then $\phi_2^{R_2}(e_1; x, e_2) \equiv \neg R_2(x, e_1, e_2)$ and it remains to update the relation $R_2$. Therefore we maintain a relation $R_3$ that contains the result of $\psi_3 = \neg (E(x, x) \rightarrow E(x, y))$ for every choice $a_1, a_2$ for $x$ and $y$ (and possibly deleted) edges $e_1, e_2, e_3$. Then
\[
\phi_3^{R_3}((e_1; x, e_2, e_3)) \equiv \exists y R_3(x, y, e_1, e_2, e_3)
\]
and it remains to update relation $R_3$ via
\[
\phi_3^{R_3}(e_1; x, y, e_2, e_3, e_4) \equiv \neg (E'(x, x, e_1, \ldots, e_4) \rightarrow E'(x, y, e_1, \ldots, e_4))
\]
where $E'$ is the edge relation obtained from $E$ by deleting $e_1, e_2, e_3$ and $e_4$, that is $E'(x, y, e_1, \ldots, e_4)$ can be replaced by
\[
E(x, y) \wedge (x, y) \neq e_1 \wedge \ldots \wedge (x, y) \neq e_4.
\]
This completes the description of the program $P$ for $\varphi$ which is easily seen to be non-recursive.

Alternatively, the proof of (a)⇒(c) can be easily adapted to show (a)⇒(d).

For simplicity we write $R_1$ instead of $R_{\varphi_1}$.

The proof of the implication (b)⇒(a) of Theorem 5.1 is based on the squirrel technique with the following idea. Given a non-recursive dynamic DY\text{N}FO-program $P = (P, \text{Init}, Q)$, we construct (again), for every update pattern $\delta = \delta_1, \ldots, \delta_j$ and every auxiliary relation $R$, a first-order formula $\varphi_{\delta}^P$ that "precomputes" the state of $R$ for every possible update sequence with the pattern $\delta$. Thanks to non-recursiveness, the formula $\varphi_{\delta}^P$ can only use relations from the input schema, once $\delta$ is longer than the number of auxiliary relations, that is, it is just a first-order formula over $\tau_m$. To obtain a first-order formula for $Q$ it thus suffices to pick such a formula with an update sequence $\delta$ that does not change the structure. We now turn to the formal statement and proof of the result.

A topological sorting of a graph $(V, E)$ is a sequence $v_1, \ldots, v_n$ such that every vertex from $V$ occurs exactly once and $i < j$ for all edges $(v_i, v_j) \in E$. Every acyclic graph has a topological sorting. In particular, if $R_1, \ldots, R_m$ is a topological sorting of the dependency graph of a non-recursive dynamic program $P = (P, \text{Init}, Q)$ then update formulas for $R_1$ do only contain relation symbols from $\tau_m$. Further we can assume, without loss of generality, that $R_m = Q$.

The following definition will be useful in the proof of Lemma 5.3. For every first-order formula $\varphi$ with $k$ free variables and every sequence $\delta = \delta_1, \ldots, \delta_j$ over $\{\text{Ins, Del}\}$ let $\varphi_{\delta_1, \ldots, \delta_j}^{P}(\vec{a}, \vec{e_1}, \ldots, \vec{e_j})$ be a $(k + 2j)$-ary formula such that for every graph $G = (V, E)$, every $\vec{a} \in V^k$ and every instantiation $\alpha = \delta_1(\vec{e_1}) \cdots \delta_j(\vec{e_j})$ of $\delta$ with tuples $\vec{e_1}, \ldots, \vec{e_j} \in V^j$:
\[
\alpha(G) \models \varphi \text{ if and only if } G \models \varphi_{\delta_1, \ldots, \delta_j}(\vec{a}, \vec{e_1}, \ldots, \vec{e_j}).
\]
It is straightforward to construct $\varphi_{\delta_1, \ldots, \delta_j}^{P}$.

**Lemma 5.3.** If a query can be maintained in non-recursive DY\text{N}FO, then it can be expressed in FO.

**Proof.** Let $Q$ be a query which can be maintained by a non-recursive DY\text{N}FO-program $P = (P, \text{Init}, Q)$ over schema $\tau = \tau_m \cup \tau_{\text{ins}}$. We assume for simplicity that $\tau_m = \{E\}$, for a binary symbol $E$. We let $R_0 \equiv E$ and assume that the auxiliary relations $R_1, \ldots, R_m$ are enumerated with respect to a topological sorting of the dependency graph of $P$ with $R_m = Q$.

We define inductively, by $i$, for every sequence $\delta_1, \ldots, \delta_j$ with $j \geq i$, first-order formulas $\varphi_{\delta_1, \ldots, \delta_j}^{R_i}(\vec{y}, \vec{x_1}, \ldots, \vec{x_j})$ over schema $\tau_m = \{E\}$ such that $\varphi_{\delta_1, \ldots, \delta_j}^{R_i}$ defines $R_i$ after updates $\delta_1(\vec{x_1}) \cdots \delta_j(\vec{x_j})$. More precisely $\varphi_{\delta_1, \ldots, \delta_j}^{R_0}$ will be defined such that for every state $S = (V, E^S, A^S)$ of $P$ and every sequence $\delta = \delta_1(\vec{a_1}) \cdots \delta_j(\vec{a_j})$ of updates the following holds:
\[
P_S(S) \shortmid R_i = \{ \vec{b} \mid (V, E) \models \varphi_{\delta_1, \ldots, \delta_j}(\vec{b}, \vec{a_1}, \ldots, \vec{a_j}) \}.
\]
For $R_0 = E$ the formula $\varphi_{\delta_1, \ldots, \delta_j}^{E}$ can be defined just as in the previous lemma. For $R_i$ with $i \geq 1$ the formula $\varphi_{\delta_1, \ldots, \delta_j}^{R_i}(\vec{y}, \vec{x_1}, \ldots, \vec{x_j})$ is obtained from the update formula $\varphi_{\delta_1, \ldots, \delta_j}^{R_{i-1}}(\vec{z}; \vec{y})$ of $R_i$ by substituting all occurrences of $R_{i-1}(\vec{z})$ by $\varphi_{\delta_1, \ldots, \delta_j}^{R_{i-1}}(\vec{x_{i-1}}, \ldots, \vec{x_1}, \vec{z})$ for all $i' < i$. Using induction over $i$, one can prove that the formulas $\varphi_{\delta_1, \ldots, \delta_j}^{R_i}$ satisfy Equation 2. As $P$ is non-recursive, each formula $\varphi_{\delta_1, \ldots, \delta_j}^{R_i}$ with $j \geq i$ over schema $\{E\}$ can be constructed as follows. The formula "guesses" a tuple
\( a \in E \), deletes and inserts it \( m \) times and applies \( \varphi^R_{\text{del ins}}(m) \) to the result (which is identical to the current graph), or (for the case that \( E \) is empty) it guesses a tuple \( \vec{a} \not\in E \), inserts and deletes it \( m \) times and applies \( \varphi^I_{\text{del ins}}(m) \) to the result.

More precisely, \( \varphi \) for \( \mathcal{Q} \) is defined by

\[
\varphi(\vec{y}) \overset{\text{def}}{=} \exists \vec{x}( (E(\vec{x}) \land \varphi^R_{\text{del ins}}(m)(\vec{y}, \vec{x}, \vec{x}, \ldots, \vec{x})) \\
\lor (\neg E(\vec{x}) \land \varphi^I_{\text{del ins}}(m)(\vec{y}, \vec{x}, \vec{x}, \ldots, \vec{x})) ).
\]

6. DISCUSSION AND FUTURE WORK

In this work, we studied dynamic conjunctive queries. We have shown that, contrary to the static setting, many fragments collapse in the dynamic world. Furthermore, a close connection between absolute semantics and \( \Delta \)-semantics for conjunctive queries has been established. These results were summarized in Figure 1. Finally, it has been shown that dynamic conjunctive queries with negations capture (static) first-order logic.

All results are for arbitrary initialization mappings. However, they also hold in the setting with first-order definable initialization mappings. They do not carry over when the initialization mapping and updates have to be definable in the same class.

Some first steps towards separation of the remaining classes have been taken. We only state the results; the proofs will appear in the full version of this work.

**Theorem 6.1.** The class \( \text{DynPropCQ} \) is a strict subclass of \( \text{DynProp} \).

**Theorem 6.2.** The class \( \text{DynProp} \) is a strict subclass of \( \text{DynCQ} \).

The first result requires some work. The second result relies on the observation that \( \text{DynCQ} \) captures the dynamic class \( \text{DynQF} \), that is, the extension of \( \text{DynProp} \) by auxiliaries functions; and the separation of \( \text{DynQF} \) and \( \text{DynProp} \) [9]. The even weaker dynamic class \( \text{DynProjections} \) (where update formulas are restricted to be projections) was separated from \( \text{DynProp} \) already in [12].

Whether the remaining classes \( \text{DynCQ} \), \( \text{DynCQ}^- \) and \( \text{DynFO} \) can be separated or collapsed remains open.

In addition to untangling the remaining variations of conjunctive queries, the dynamic quantifier hierarchy and quantifier alternation hierarchy, respectively, deserve a closer look. Lemma 3.9 shows that in the dynamic setting the \( \Sigma_i \) and \( \Pi_i \)-fragment of first-order logic coincide. Whether there is a strict \( \Sigma_i \)-hierarchy remains open. Furthermore, the equivalence of \( \exists \Sigma^i \text{FO} \) with absolute and \( \Delta \)-semantics does not immediately translate to fragments of \( \text{FO} \) with alternating quantifiers (although one of the direction does, see Lemma 4.6).

Capturing first-order logic by dynamic conjunctive queries with negations does not immediately yield performance gains (since a first-order queries with \( k \) quantifiers is translated to a dynamic \( \text{DynCQ}^- \)-program of arity at least \( k \)). In future work we plan to study whether the work that has been started here can be used to improve the performance of query maintenance.

7. REFERENCES


