Answering UCQs under Updates and in the Presence of Integrity Constraints

Christoph Berkholz
Humboldt-Universität zu Berlin, Germany
berkholz@informatik.hu-berlin.de

Jens Keppeler
Humboldt-Universität zu Berlin, Germany
keppelej@informatik.hu-berlin.de

Nicole Schweikardt
Humboldt-Universität zu Berlin, Germany
schweikn@informatik.hu-berlin.de

Abstract

We investigate the query evaluation problem for fixed queries over fully dynamic databases where tuples can be inserted or deleted. The task is to design a dynamic data structure that can immediately report the new result of a fixed query after every database update. We consider unions of conjunctive queries (UCQs) and focus on the query evaluation tasks *testing* (decide whether an input tuple $\overline{a}$ belongs to the query result), *enumeration* (enumerate, without repetition, all tuples in the query result), and *counting* (output the number of tuples in the query result).

We identify three increasingly restrictive classes of UCQs which we call *t-hierarchical*, *q-hierarchical*, and *exhaustively q-hierarchical* UCQs. Our main results provide the following dichotomies: If the query's homomorphic core is t-hierarchical (q-hierarchical, exhaustively q-hierarchical), then the testing (enumeration, counting) problem can be solved with constant update time and constant testing time (delay, counting time). Otherwise, it cannot be solved with sublinear update time and sublinear testing time (delay, counting time), unless the OV-conjecture and/or the OMv-conjecture fails.

We also study the complexity of query evaluation in the dynamic setting in the presence of integrity constraints, and we obtain similar dichotomy results for the special case of small domain constraints (i.e., constraints which state that all values in a particular column of a relation belong to a fixed domain of constant size).

2012 ACM Subject Classification Theory of computation → Database query languages (principles), Theory of computation → Logic and databases

Keywords and phrases dynamic query evaluation, union of conjunctive queries, constant-delay enumeration, counting problem, testing

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.8

Funding Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SCHW 837/5-1

Acknowledgements We are very grateful to the anonymous reviewers — their valuable feedback helped to significantly improve the paper. We also acknowledge the financial support by the German Research Foundation DFG under grant SCHW 837/5-1.

21st International Conference on Database Theory (ICDT 2018).
Editors: Benny Kimelfeld and Yael Amsterdamer; Article No. 8; pp. 8:1–8:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Dynamic query evaluation refers to a setting where a fixed query q has to be evaluated against a database that is constantly updated [20]. In this paper, we study dynamic query evaluation for unions of conjunctive queries (UCQs) on relational databases that may be updated by inserting or deleting tuples. A dynamic algorithm for evaluating a query q receives an initial database and performs a preprocessing phase which builds a data structure that contains a suitable representation of the database and the result of q on this database. After every database update, the data structure is updated so that it suitably represents the new database D and the result $q(D)$ of q on this database.

To solve the *counting problem*, such an algorithm is required to quickly report the number $|q(D)|$ of tuples in the current query result, and the *counting time* is the time used to compute this number. To solve the *testing problem*, the algorithm has to be able to check for an arbitrary input tuple if it belongs to the current query result, and the *testing time* is the time used to perform this check. To solve the *enumeration problem*, the algorithm has to enumerate $q(D)$ without repetition and with a bounded *delay* between the output tuples. The *update time* is the time used for updating the data structure after having received a database update. We regard the counting (testing, enumeration) problem of a query q to be *tractable under updates* if it can be solved by a dynamic algorithm with linear preprocessing time, constant update time, and constant counting time (testing time, delay).

This setting has been studied for conjunctive queries (CQs) in our previous paper [5], which identified a class of CQs called *q-hierarchical* that precisely characterises the tractability frontier of the counting problem and the enumeration problem for CQs under updates: For every q-hierarchical CQ, the counting problem and the enumeration problem can be solved with linear preprocessing time, constant update time, constant counting time, and constant delay. And for every CQ that is not equivalent to a q-hierarchical CQ, the counting problem (and for the case of self-join-free queries, the enumeration problem) cannot be solved with sublinear update time and sublinear counting time (delay), unless the OMv-conjecture or the OV-conjecture (the OMv-conjecture) fails. The latter are well-known algorithmic conjectures on the hardness of the Boolean online matrix-vector multiplication problem (OMv) and the Boolean orthogonal vectors problem (OV) [19, 1], and "sublinear" means $O(n^{1-\epsilon})$, where $\epsilon > 0$ and n is the size of the active domain of the current database.

Our contribution. We identify a new subclass of CQs which we call *t-hierarchical*, which contains and properly extends the class of q-hierarchical CQs, and which precisely characterises the tractability frontier of the testing problem for CQs under updates (see Theorem 3.4): For every t-hierarchical CQ, the testing problem can be solved by a dynamic algorithm with linear preprocessing time, constant update time, and constant testing time. And for every CQ that is not equivalent to a t-hierarchical CQ, the testing problem cannot be solved with arbitrary preprocessing time, sublinear update time, and sublinear testing time, unless the OMv-conjecture fails.

Furthermore, we transfer the notions of t-hierarchical and q-hierarchical queries to unions of conjunctive queries (UCQs) and identify a further class of UCQs which we call *exhaustively q-hierarchical*, yielding three increasingly restricted subclasses of UCQs. In a nutshell, our main contribution concerning UCQs shows that these notions precisely characterise the tractability frontiers of the testing problem, the enumeration problem, and the counting problem for UCQs under updates (see the Theorems 4.1, 4.2, 4.5): For every t-hierarchical (q-hierarchical, exhaustively q-hierarchical) UCQ, the testing (enumeration, counting) problem can be solved with linear preprocessing time, constant update time, and constant testing time (delay, counting time). And for every UCQ that is not equivalent to a t-hierarchical (q-hierarchical, exhaustively q-hierarchical) UCQ, the testing (enumeration, counting) problem cannot be solved with sublinear update time and sublinear testing time (delay, counting time). To be precise, the lower bound for enumeration is obtained only for self-join-free queries, the lower bounds for testing and enumeration are conditioned on the OMv-conjecture, and the lower bound for counting is conditioned on the OMv-conjecture and the OV-conjecture.

Finally, we transfer our results to a scenario where databases are required to satisfy a set of *small domain constraints* (i.e., constraints stating that all values which occur in a particular column of a relation belong to a fixed domain of constant size), leading to a precise characterisation of the UCQs for which the testing (enumeration, counting) problem under updates is tractable in this scenario (see Theorem 5.3).

Further related work. The complexity of evaluating CQs and UCQs in the *static* setting (i.e., without database updates) is well-studied. In particular, there are characterisations of "tractable" queries known for Boolean queries [17, 16, 24] as well as for the task of counting the result tuples [12, 8, 13, 15, 9]. In [3], the fragment of self-join-free CQs that can be enumerated with constant delay after linear preprocessing time has been identified, but almost nothing is known about the complexity of the enumeration problem for UCQs on static databases. Very recent papers also studied the complexity of CQs with respect to a given set of integrity constraints [14, 21, 4]. The *dynamic* query evaluation problem has been considered from different angles, including *descriptive dynamic complexity* [27, 28, 29] and, somewhat closer to what we are aiming for, *incremental view maintenance* [18, 10, 22, 23, 26]. In [20], the enumeration and testing problem under updates has been studied for q-hierarchical and (more general) acyclic CQs in a setting that is very similar to our setting and the setting of [5]; the *Dynamic Constant-delay Linear Representations* (DCLR) of [20] are data structures that use at most linear update time and solve the enumeration problem and the testing problem with constant delay and constant testing time.

Outline. The rest of the paper is structured as follows. Section 2 provides basic notations concerning databases, queries, and dynamic algorithms for query evaluation. Section 3 is devoted to CQs and proves our dichotomy result concerning the testing problem for CQs. Section 4 focuses on UCQs and proves our dichotomies concerning the testing, enumeration, and counting problem for UCQs. Section 5 is devoted to integrity constraints. Due to space restrictions, some proof details had to be deferred to the paper's full version [6].

2 Preliminaries

Basic notation. We write $\mathbb{N}$ for the set of non-negative integers and let $\mathbb{N}_{\geqslant 1} := \mathbb{N} \setminus \{0\}$ and $[n] := \{1, \dots, n\}$ for all $n \in \mathbb{N}_{\geqslant 1}$. By 2^S we denote the power set of a set S. We write $\vec{v}_i$ to denote the i-th component of an n-dimensional vector $\vec{v}$, and we write $M_{i,j}$ for the entry in row i and column j of a matrix M. By $()$ we denote the empty tuple, i.e., the unique tuple of arity 0. For an r-tuple $t = (t_1, \dots, t_r)$ and indices $i_1, \dots, i_m \in \{1, \dots, r\}$ we write $\pi_{i_1,\dots,i_m}(t)$ to denote the projection of t to the components $i_1, \dots, i_m$, i.e., the m-tuple $(t_{i_1}, \dots, t_{i_m})$, and in case that $m = 1$ we identify the 1-tuple (t_{i_1}) with the element t_{i_1}. For a set T of r-tuples we let $\pi_{i_1,\dots,i_m}(T) := \{\pi_{i_1,\dots,i_m}(t) \ : \ t \in T\}$.

Databases. We fix a countably infinite set $\mathbf{dom}$, the *domain* of potential database entries. Elements in $\mathbf{dom}$ are called *constants*. A *schema* is a finite set σ of relation symbols, where each $R \in \sigma$ is equipped with a fixed *arity* $\operatorname{ar}(R) \in \mathbb{N}$ (note that here we explicitly allow relation symbols of arity 0). Let us fix a schema $\sigma = \{R_1, \ldots, R_s\}$, and let $r_i := \operatorname{ar}(R_i)$ for $i \in [s]$. A *database* D of schema σ (σ-db, for short), is of the form $D = (R_1^D, \ldots, R_s^D)$, where R_i^D is a finite subset of $\mathbf{dom}^{r_i}$. The *active domain* $\operatorname{adom}(D)$ of D is the smallest subset A of $\mathbf{dom}$ such that $R_i^D \subseteq A^{r_i}$ for all $i \in [s]$.

Queries. We fix a countably infinite set $\mathbf{var}$ of *variables*. We allow queries to use variables and constants. An *atom* ψ of schema σ is of the form $Rv_1 \cdots v_r$ with $R \in \sigma$, $r = \operatorname{ar}(R)$, and $v_1, \ldots, v_r \in \mathbf{var} \cup \mathbf{dom}$. A *conjunctive formula* of schema σ is of the form

$$\exists y_1 \cdots \exists y_\ell \, \big(\psi_1 \wedge \cdots \wedge \psi_d \big) \tag{$*$}$$

where $\ell \geqslant 0$, $d \geqslant 1$, ψ_j is an atom of schema σ for every $j \in [d]$, and $y_1, \ldots, y_\ell$ are distinct elements in $\mathbf{var}$. For a conjunctive formula φ of the form $(*)$ we let $\operatorname{vars}(\varphi)$ (and $\operatorname{cons}(\varphi)$, respectively) be the set of all variables (and constants, respectively) occurring in φ. The set of *free* variables of φ is $\operatorname{free}(\varphi) := \operatorname{vars}(\varphi) \setminus \{y_1, \ldots, y_\ell\}$. For every variable $x \in \operatorname{vars}(\varphi)$ we let $\operatorname{atoms}_\varphi(x)$ (or $\operatorname{atoms}(x)$, if φ is clear from the context) be the set of all atoms ψ_j of φ such that $x \in \operatorname{vars}(\psi_j)$. The formula φ is called *quantifier-free* if $\ell = 0$, and it is called *self-join-free* if no relation symbol occurs more than once in φ.

For $k \geqslant 0$, a *k-ary conjunctive query* (k-ary CQ, for short) is of the form

$$\{ \, (u_1, \ldots, u_k) \; : \; \varphi \, \} \tag{$**$}$$

where φ is a conjunctive formula of schema σ, $u_1, \ldots, u_k \in \operatorname{free}(\varphi) \cup \mathbf{dom}$, and $\{u_1, \ldots, u_k\} \cap \mathbf{var} = \operatorname{free}(\varphi)$. We often write $q_\varphi(\overline{u})$ for $\overline{u} = (u_1, \ldots, u_k)$ (or q_φ if $\overline{u}$ is clear from the context) to denote such a query. We let $\operatorname{vars}(q_\varphi) := \operatorname{vars}(\varphi)$, $\operatorname{free}(q_\varphi) := \operatorname{free}(\varphi)$, and $\operatorname{cons}(q_\varphi) := \operatorname{cons}(\varphi) \cup (\{u_1, \ldots, u_k\} \cap \mathbf{dom})$. For every $x \in \operatorname{vars}(q_\varphi)$ we let $\operatorname{atoms}_{q_\varphi}(x) := \operatorname{atoms}_\varphi(x)$, and if q_φ is clear from the context, we omit the subscript and simply write $\operatorname{atoms}(x)$. The CQ q_φ is called *quantifier-free* (*self-join-free*) if φ is quantifier-free (self-join-free).

The semantics are defined as usual: A *valuation* is a mapping $\beta : \operatorname{vars}(q_\varphi) \cup \mathbf{dom} \to \mathbf{dom}$ with $\beta(a) = a$ for every $a \in \mathbf{dom}$. A valuation β is a *homomorphism* from q_φ to a σ-db D if for every atom $Rv_1 \cdots v_r$ in q_φ we have $\big(\beta(v_1), \ldots, \beta(v_r)\big) \in R^D$. We sometimes write $\beta : q_\varphi \to D$ to indicate that β is a homomorphism from q_φ to D. The *query result* $q_\varphi(D)$ of a k-ary CQ $q_\varphi(u_1, \ldots, u_k)$ on the σ-db D is defined as the set $\{ \, \big(\beta(u_1), \ldots, \beta(u_k)\big) \; : \; \beta$ is a homomorphism from q_φ to $D\}$. If $\overline{x} = (x_1, \ldots, x_k)$ is a list of the free variables of φ and $\overline{a} \in \mathbf{dom}^k$, we sometimes write $D \models \varphi[\overline{a}]$ to indicate that there is a homomorphism $\beta : q \to D$ with $\overline{a} = \big(\beta(x_1), \ldots, \beta(x_k)\big)$, for the query $q = q_\varphi(x_1, \ldots, x_k)$.

A *k-ary union of conjunctive queries* (UCQ) is of the form $q_1(\overline{u}_1) \cup \cdots \cup q_d(\overline{u}_d)$ where $d \geqslant 1$ and $q_i(\overline{u}_i)$ is a k-ary CQ of schema σ for every $i \in [d]$. The query result of such a UCQ q on a σ-db D is $q(D) := \bigcup_{i=1}^d q_i(D)$. For example, $\{(x, y) : Exy\} \cup \{(x, x) : Exy\} \cup \{(y, y) : Exy\}$ is a 2-ary UCQ q with $q(D) = E^D \cup \{ \, (a, a) \; : \; a \in \operatorname{adom}(D) \, \}$ for every $\{E\}$-db D.

For a k-ary query q we write $\operatorname{vars}(q)$ (and $\operatorname{cons}(q)$) to denote the set of all variables (and constants) that occur in q. Clearly, $q(D) \subseteq (\operatorname{adom}(D) \cup \operatorname{cons}(q))^k$. A *Boolean* query is a query of arity $k = 0$. As usual, for Boolean queries q we will write $q(D) = \texttt{yes}$ instead of $q(D) \neq \emptyset$, and $q(D) = \texttt{no}$ instead of $q(D) = \emptyset$. Two k-ary queries q and q' are *equivalent* ($q \equiv q'$, for short) if $q(D) = q'(D)$ for every σ-db D.

Homomorphisms. We use standard notation concerning homomorphisms (cf., e.g. [2]). The notion of a homomorphism $\beta : q \to D$ from a CQ q to a database D has already been defined above. A homomorphism $g : D \to q$ from a database D to a CQ q is a mapping from $\text{adom}(D)$ to $\text{vars}(q) \cup \text{cons}(q)$ such that $g(a) = a$ for all $a \in \text{adom}(D) \cap \text{cons}(q)$ and whenever $(a_1, \dots, a_r)$ is a tuple in some relation R^D of D, then $Rg(a_1) \cdots g(a_r)$ is an atom of q.

Let $q(u_1, \dots, u_k)$ and $q'(v_1, \dots, v_k)$ be two k-ary CQs. A *homomorphism* from q to q' is a mapping $h\colon \text{vars}(q) \cup \mathbf{dom} \to \text{vars}(q') \cup \mathbf{dom}$ with $h(a) = a$ for all $a \in \mathbf{dom}$ and $h(u_i) = v_i$ for all $i \in [k]$ such that for every atom $Rw_1 \cdots w_r$ in q there is an atom $Rh(w_1) \cdots h(w_r)$ in q'. We sometimes write $h : q \to q'$ to indicate that h is a homomorphism from q to q'. Note that by [7] there is a homomorphism from q to q' if and only if for every database D it holds that $q(D) \supseteq q'(D)$. A CQ q is a *homomorphic core* if there is no homomorphism from q into a proper subquery of q. Here, a *subquery* of a CQ $q_\varphi(\overline{u})$ where φ is of the form $(*)$ is a CQ $q_{\varphi'}(\overline{u})$ where φ' is of the form $\exists y_{i_1} \cdots \exists y_{i_m} \ (\psi_{j_1} \wedge \cdots \wedge \psi_{j_n})$ with $i_1, \dots, i_m \in [\ell]$, $j_1, \dots, j_n \in [d]$, and $\text{free}(\varphi') = \text{free}(\varphi)$.

We say that a UCQ is a *homomorphic core* if every CQ in the union is a homomorphic core and there is no homomorphism between two distinct CQs. It is well-known that every CQ and every UCQ is equivalent to a unique (up to renaming of variables) homomorphic core, which is therefore called *the core of* the query (cf., e.g., [2]).

Sizes and Cardinalities. The *size* $\|\sigma\|$ of a schema σ is $|\sigma| + \sum_{R \in \sigma} \text{ar}(R)$. The size $\|q\|$ of a query q of schema σ is the length of q when viewed as a word over the alphabet $\sigma \cup \mathbf{var} \cup \mathbf{dom} \cup \{\wedge, \exists, (,), \{,\}, :, \cup\} \cup \{,\}$. For a k-ary query q and a σ-db D, the *cardinality of the query result* is the number $|q(D)|$ of tuples in $q(D)$. The *cardinality* $|D|$ of a σ-db D is defined as the number of tuples stored in D, i.e., $|D| := \sum_{R \in \sigma} |R^D|$. The *size* $\|D\|$ of D is defined as $\|\sigma\| + |\text{adom}(D)| + \sum_{R \in \sigma} \text{ar}(R) \cdot |R^D|$ and corresponds to the size of a reasonable encoding of D.

The following notions concerning updates, dynamic algorithms for query evaluation, and algorithmic conjectures are taken almost verbatim from [5].

Updates. We allow to update a σ-db by inserting or deleting tuples as follows. An *insertion* command is of the form insert $R(a_1, \dots, a_r)$ for $R \in \sigma$, $r = \text{ar}(R)$, and $a_1, \dots, a_r \in \mathbf{dom}$. When applied to a σ-db D, it results in the updated σ-db D' with $R^{D'} := R^D \cup \{(a_1, \dots, a_r)\}$ and $S^{D'} := S^D$ for all $S \in \sigma \setminus \{R\}$. A *deletion* command is of the form delete $R(a_1, \dots, a_r)$ for $R \in \sigma$, $r = \text{ar}(R)$, and $a_1, \dots, a_r \in \mathbf{dom}$. When applied to a σ-db D, it results in the updated σ-db D' with $R^{D'} := R^D \setminus \{(a_1, \dots, a_r)\}$ and $S^{D'} := S^D$ for all $S \in \sigma \setminus \{R\}$. Note that both types of commands may change the database's active domain.

Dynamic algorithms for query evaluation. Following [11], we use Random Access Machines (RAMs) with $O(\log n)$ word-size and a uniform cost measure to analyse our algorithms. We will assume that the RAM's memory is initialised to 0. In particular, if an algorithm uses an array, we will assume that all array entries are initialised to 0, and this initialisation comes at no cost (in real-world computers this can be achieved by using the *lazy array initialisation technique*, cf. [25]). A further assumption is that for every fixed dimension $k \in \mathbb{N}_{\geqslant 1}$ we have available an unbounded number of k-ary arrays $\mathtt{A}$ such that for given $(n_1, \dots, n_k) \in \mathbb{N}^k$ the entry $\mathtt{A}[n_1, \dots, n_k]$ at position $(n_1, \dots, n_k)$ can be accessed in constant time (while this can be accomplished easily in the RAM-model, for an implementation on real-world computers one would probably have to resort to replacing our use of arrays by using suitably designed hash functions). For our purposes it will be convenient to assume that $\mathbf{dom} = \mathbb{N}_{\geqslant 1}$.

Our algorithms will take as input a k-ary query q and a σ-db D_0. For all query evaluation problems considered in this paper, we aim at routines **preprocess** and **update** which achieve the following. Upon input of q and D_0, the **preprocess** routine builds a data structure $\texttt{D}$ which represents D_0 (and which is designed in such a way that it supports the evaluation of q on D_0). Upon input of a command $\texttt{update}\ R(a_1, \ldots, a_r)$ (with $\texttt{update} \in \{\mathsf{insert}, \mathsf{delete}\}$), calling **update** modifies the data structure $\texttt{D}$ such that it represents the updated database D. The *preprocessing time* t_p is the time used for performing **preprocess**. The *update time* t_u is the time used for performing an **update**, and in this paper we aim at algorithms where t_u is independent of the size of the current database D. By **init** we denote the particular case of the routine **preprocess** upon input of a query q and the *empty* database $D_\emptyset$, where $R^{D_\emptyset} = \emptyset$ for all $R \in \sigma$. The *initialisation time* t_i is the time used for performing **init**. In all algorithms presented in this paper, the **preprocess** routine for input of q and D_0 will carry out the **init** routine for q and then perform a sequence of $|D_0|$ update operations to insert all the tuples of D_0 into the data structure. Consequently, $t_p = t_i + |D_0| \cdot t_u$.

In the following, D will always denote the database that is currently represented by the data structure $\texttt{D}$. To solve the *enumeration problem under updates*, apart from the routines **preprocess** and **update**, we aim at a routine **enumerate** such that calling **enumerate** invokes an enumeration of all tuples, *without repetition*, that belong to the query result $q(D)$. The *delay* t_d is the maximum time used during a call of **enumerate**

- until the output of the first tuple (or the end-of-enumeration message $\texttt{EOE}$, if $q(D) = \emptyset$),
- between the output of two consecutive tuples, and
- between the output of the last tuple and the end-of-enumeration message $\texttt{EOE}$.

To *test* if a given tuple belongs to the query result, instead of **enumerate** we aim at a routine **test** which upon input of a tuple $\overline{a} \in \mathbf{dom}^k$ checks whether $\overline{a} \in q(D)$. The *testing time* t_t is the time used for performing a **test**. To solve the *counting problem under updates*, we aim at a routine **count** which outputs the cardinality $|q(D)|$ of the query result. The *counting time* t_c is the time used for performing a **count**. To *answer* a *Boolean* query under updates, we aim at a routine **answer** that produces the answer $\texttt{yes}$ or $\texttt{no}$ of q on D. The *answer time* t_a is the time used for performing **answer**. Whenever speaking of a *dynamic algorithm*, we mean an algorithm that has routines **preprocess** and **update** and, depending on the problem at hand, at least one of the routines **answer**, **test**, **count**, and **enumerate**.

When writing *poly*(n) we mean $n^{O(1)}$, and for a query q we often write *poly*(q) instead of *poly*$(\|q\|)$. We will often adopt the view of *data complexity* and suppress factors that may depend on the query q but not on the database D. E.g., "linear preprocessing time" means $t_p \leqslant f(q) \cdot \|D_0\|$ and "constant update time" means $t_u \leqslant f(q)$, for some function f.

Algorithmic conjectures. Similarly to [5] we obtain hardness results that are conditioned on algorithmic conjectures concerning the hardness of the following problems. These problems deal with *Boolean* matrices and vectors, i.e., matrices and vectors over $\{0, 1\}$, and all the arithmetic is done over the Boolean semiring, where multiplication means conjunction and addition means disjunction.

The *orthogonal vectors problem* (OV-problem) is the following decision problem. Given two sets U and V of n Boolean vectors of dimension d, decide whether there are vectors $\vec{u} \in U$ and $\vec{v} \in V$ such that $\vec{u}^{\mathsf{T}} \vec{v} = 0$. The *OV-conjecture* states that there is no $\epsilon > 0$ such that the OV-problem for $d = \lceil \log^2 n \rceil$ can be solved in time $O(n^{2-\epsilon})$, see [1].

The *online matrix-vector multiplication problem* (OMv-problem) is the following algorithmic task. At first, the algorithm gets a Boolean $n \times n$ matrix M and is allowed to do some preprocessing. Afterwards, the algorithm receives n vectors $\vec{v}^{\,1}, \ldots, \vec{v}^{\,n}$ one by one and has to output $M\vec{v}^t$ before it has access to $\vec{v}^{t+1}$ (for each $t < n$). The running time is the overall time the algorithm needs to produce the output $M\vec{v}^1, \dots, M\vec{v}^n$. The *OMv-conjecture* [19] states that there is no $\epsilon > 0$ such that the OMv-problem can be solved in time $O(n^{3-\epsilon})$.

A related problem is the *OuMv-problem* where the algorithm, again, is given a Boolean $n \times n$ matrix M and is allowed to do some preprocessing. Afterwards, the algorithm receives a sequence of pairs of n-dimensional Boolean vectors $\vec{u}^t, \vec{v}^t$ for each $t \in [n]$, and the task is to compute $(\vec{u}^t)^\top M\vec{v}^t$ before accessing $\vec{u}^{t+1}, \vec{v}^{t+1}$. The *OuMv-conjecture* states that there is no $\epsilon > 0$ such that the OuMv-problem can be solved in time $O(n^{3-\epsilon})$. It was shown in [19] that the OuMv-conjecture is equivalent to the OMv-conjecture, i.e., the OuMv-conjecture fails if, and only if, the OMv-conjecture fails.

3 Conjunctive queries

This section's aim is twofold: Firstly, we observe that the notions and results of [5] generalise to CQs with constants in a straightforward way. Secondly, we identify a new subclass of CQs which precisely characterises the CQs for which *testing* can be done efficiently under updates.

The definition of *q-hierarchical* CQs can be taken verbatim from [5]:

▶ **Definition 3.1.** A CQ q is *q-hierarchical* if for any two variables $x, y \in \text{vars}(q)$ we have

(i) $\text{atoms}(x) \subseteq \text{atoms}(y)$ or $\text{atoms}(y) \subseteq \text{atoms}(x)$ or $\text{atoms}(x) \cap \text{atoms}(y) = \emptyset$, and

(ii) if $\text{atoms}(x) \subsetneq \text{atoms}(y)$ and $x \in \text{free}(q)$, then $y \in \text{free}(q)$.

Obviously, it can be checked in time *poly*(q) whether a given CQ q is q-hierarchical. It is straightforward to see that if a CQ is q-hierarchical, then so is its homomorphic core. In particular, a CQ is equivalent to a q-hierarchical CQ iff its homomorphic core is q-hierarchical. Using the main results of [5], it is not difficult to show the following; for details see [6].

▶ **Theorem 3.2.**

(a) *There is a dynamic algorithm that receives a q-hierarchical k-ary CQ q and a σ-db D_0, and computes within $t_p = poly(q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(q)$ and allows to*

- **(i)** *compute the cardinality $|q(D)|$ in time $t_c = O(1)$,*
- **(ii)** *enumerate $q(D)$ with delay $t_d = poly(q)$,*
- **(iii)** *test for an input tuple $\overline{a} \in \textbf{dom}^k$ if $\overline{a} \in q(D)$ within time $t_t = poly(q)$,*
- **(iv)** *and when given a tuple $\overline{a} \in q(D)$, the tuple $\overline{a}'$ (or the message* EOE*) that the enumeration procedure of* (aii) *would output directly after having output $\overline{a}$, can be computed within time poly(q).*

(b) *Let $\epsilon > 0$ and let q be a CQ that is not equivalent to a q-hierarchical CQ.*

- **(i)** *If q is Boolean, then there is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that answers $q(D)$ in time $t_a = O(n^{2-\varepsilon})$, unless the OMv-conjecture fails.*
- **(ii)** *There is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that computes the cardinality $|q(D)|$ in time $t_c = O(n^{1-\varepsilon})$, unless the OMv-conjecture or the OV-conjecture fails.*
- **(iii)** *If q is self-join-free, then there is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that enumerates $q(D)$ with delay $t_d = O(n^{1-\varepsilon})$, unless the OMv-conjecture fails.*

All lower bounds remain true if we restrict ourselves to the class of databases that map homomorphically into q.

Note that neither the results of [5] nor Theorem 3.2 provide a precise characterisation of the CQs for which *testing* can be done efficiently under updates. Of course, according to Theorem 3.2 (aiii), the testing problem can be solved with constant update time and constant testing time for every *q-hierarchical* CQ. But the same holds true for the non-q-hierarchical CQ $p_{S\text{-}E\text{-}T} := \{\, (x, y) \, : \, Sx \wedge Exy \wedge Ty \,\}$. The corresponding dynamic algorithm simply uses 1-dimensional arrays $\mathtt{A}_S$ and $\mathtt{A}_T$ and a 2-dimensional array $\mathtt{A}_E$ such that for all $a, b \in \mathbf{dom}$ we have $\mathtt{A}_E[a, b] = 1$ if $(a, b) \in E^D$, and $\mathtt{A}_E[a, b] = 0$ otherwise, and $\mathtt{A}_R[a] = 1$ if $a \in R^D$, and $\mathtt{A}_R[a] = 0$ otherwise, for $R \in \{S, T\}$. When given an update command, the arrays can be updated within constant time. And when given a tuple $(a, b) \in \mathbf{dom}^2$, the **test** routine simply looks up the array entries $\mathtt{A}_S[a]$, $\mathtt{A}_E[a, b]$, $\mathtt{A}_T[b]$ and returns the correct query result accordingly. To characterise the conjunctive queries for which testing can be done efficiently under updates, we introduce the following notion of *t-hierarchical* CQs.

▶ **Definition 3.3.** A CQ q is *t-hierarchical* if the following is satisfied:

(i) for all $x, y \in \text{vars}(q) \setminus \text{free}(q)$, we have
$\text{atoms}(x) \subseteq \text{atoms}(y)$ or $\text{atoms}(y) \subseteq \text{atoms}(x)$ or $\text{atoms}(x) \cap \text{atoms}(y) = \emptyset$, and

(ii) for all $x \in \text{free}(q)$ and all $y \in \text{vars}(q) \setminus \text{free}(q)$, we have
$\text{atoms}(x) \cap \text{atoms}(y) = \emptyset$ or $\text{atoms}(y) \subseteq \text{atoms}(x)$.

Obviously, it can be checked in time *poly*(q) whether a given CQ q is t-hierarchical. Note that every q-hierarchical CQ is t-hierarchical, and a *Boolean* query is t-hierarchical if and only if it is q-hierarchical. The queries $p_{S\text{-}E\text{-}T}$ and $p_{E\text{-}E\text{-}R} := \{\, (x, y) \, : \, \exists v_1 \exists v_2 \exists v_3 \, (\, Exv_1 \wedge Eyv_2 \wedge Rxyv_3 \,) \,\}$ are examples for queries that are t-hierarchical but not q-hierarchical. It is straightforward to verify that if a CQ is t-hierarchical, then so is its homomorphic core. This section's main result shows that the *t-hierarchical* CQs precisely characterise the CQs for which the *testing* problem can be solved efficiently under updates:

▶ **Theorem 3.4.**

(a) *There is a dynamic algorithm that receives a t-hierarchical k-ary CQ q and a σ-db D_0, and computes within $t_p = poly(q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(q)$ and allows to test for an input tuple $\overline{a} \in \boldsymbol{dom}^k$ if $\overline{a} \in q(D)$ within time $t_t = poly(q)$.*

(b) *Let $\epsilon > 0$ and let q be a k-ary CQ that is not equivalent to a t-hierarchical CQ. There is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\epsilon})$ update time that can test for any input tuple $\overline{a} \in \boldsymbol{dom}^k$ if $\overline{a} \in q(D)$ within testing time $t_t = O(n^{1-\epsilon})$, unless the OMv-conjecture fails. The lower bound remains true if we restrict ourselves to the class of databases that map homomorphically into q.*

Proof. To avoid notational clutter, and without loss of generality, we restrict attention to queries $q_\varphi(u_1, \ldots, u_k)$ where $(u_1, \ldots, u_k)$ is of the form $(z_1, \ldots, z_k)$ for pairwise distinct variables $z_1, \ldots, z_k$. For the proof of (a), we combine the array construction described above for the example query $p_{S\text{-}E\text{-}T}$ with the dynamic algorithm provided by Theorem 3.2 (a) and the following Lemma 3.5. To formulate the lemma, we need the following notation. A k-ary *generalised CQ* is of the form $\{\, (z_1, \ldots, z_k) \, : \, \varphi_1 \wedge \cdots \wedge \varphi_m \,\}$ where $k \geqslant 0$, $z_1, \ldots, z_k$ are pairwise distinct variables, $m \geqslant 1$, φ_j is a conjunctive formula for each $j \in [m]$, $\text{free}(\varphi_1) \cup \cdots \cup \text{free}(\varphi_m) = \{z_1, \ldots, z_k\}$, and the quantified variables of φ_j and $\varphi_{j'}$ are pairwise disjoint for all $j, j' \in [m]$ with $j \neq j'$ and disjoint from $\{z_1, \ldots, z_k\}$. For each $j \in [m]$ let $\overline{z}^{(j)}$ be the sublist of $\overline{z} := (z_1, \ldots, z_k)$ that only contains the variables in $\text{free}(\varphi_j)$. I.e., $\overline{z}^{(j)}$ is obtained from $\overline{z}$ by deleting all variables that do not belong to $\text{free}(\varphi_j)$. Accordingly, for a tuple $\overline{a} = (a_1, \ldots, a_k) \in \mathbf{dom}^k$ by $\overline{a}^{(j)}$ we denote the tuple that contains exactly those a_i

where z_i belongs to $\overline{z}^{(j)}$. The query result of q on a σ-db D is the set

$$q(D) \ := \ \{ \ \overline{a} \in \mathbf{dom}^k \ : \ D \models \varphi_j[\overline{a}^{(j)}] \ \text{ for each } j \in [m] \ \},$$

where $D \models \varphi_j[\overline{a}^{(j)}]$ means that there is a homomorphism $\beta_j : q_j \to D$ for the query $q_j := \{ \overline{z}^{(j)} : \varphi_j \}$, with $\beta_j(z_i) = a_i$ for every i with $z_i \in \text{free}(\varphi_j)$. For example, $p'_{E\text{-}E\text{-}R} := \{ (x, y) : \exists v_1 \, Exv_1 \wedge \exists v_2 \, Eyv_2 \wedge \exists v_3 \, Rxyv_3 \}$ is a generalised CQ that is equivalent to the CQ $p_{E\text{-}E\text{-}R}$. The proof of the following lemma can be found in the appendix.

▶ **Lemma 3.5.** *Every t-hierarchical CQ $q_\varphi(z_1, \ldots, z_k)$ is equivalent to a generalised CQ $q' = \{ (z_1, \ldots, z_k) : \varphi_1 \wedge \cdots \wedge \varphi_m \}$ such that for each $j \in [m]$ the CQ $q_j := \{ \overline{z}^{(j)} : \varphi_j \}$ is q-hierarchical or quantifier-free. Furthermore, there is an algorithm which decides in time poly(q_φ) whether q_φ is t-hierarchical, and if so, outputs an according q'.*

The proof of Theorem 3.4 (a) now follows easily: When given a t-hierarchical CQ $q_\varphi(z_1, \ldots, z_k)$, use the algorithm provided by Lemma 3.5 to compute an equivalent generalised CQ q' of the form $\{(z_1, \ldots, z_k) : \varphi_1 \wedge \cdots \wedge \varphi_m\}$ and let $q_j := \{\overline{z}^{(j)} : \varphi_j\}$ for each $j \in [m]$. W.l.o.g. assume that there is an $m' \in \{0, \ldots, m\}$ such that q_j is q-hierarchical for each $j \leqslant m'$ and q_j is quantifier-free for each $j > m'$. We use in parallel, for each $j \leqslant m'$, the data structures provided by Theorem 3.2 (a) for the q-hierarchical CQ q_j. In addition to this, we use an r-dimensional array $\mathtt{A}_R$ for each relation symbol $R \in \sigma$ of arity $r := \text{ar}(R)$, and we ensure that for all $\overline{b} \in \mathbf{dom}^r$ we have $\mathtt{A}_R[\overline{b}] = 1$ if $\overline{b} \in R^D$, and $\mathtt{A}_R[\overline{b}] = 0$ otherwise. When receiving an update command $\mathsf{update}\,R(\overline{b})$, we let $\mathtt{A}_R[\overline{b}] := 1$ if $\mathsf{update} = \mathsf{insert}$, and $\mathtt{A}_R[\overline{b}] := 0$ if $\mathsf{update} = \mathsf{delete}$, and in addition to this, we call the **update** routines of the data structure for $q^{(j)}$ for each $j \leqslant m'$. Upon input of a tuple $\overline{a} \in \mathbf{dom}^k$, the **test** routine proceeds as follows. For each $j \leqslant m'$, it calls the **test** routine of the data structure for $q^{(j)}$ upon input $\overline{a}^{(j)}$. Additionally, it uses the arrays $\mathtt{A}_R$ for all $R \in \sigma$ to check if for each $j > m'$ the quantifier-free query $q^{(j)}$ is satisfied by the tuple $\overline{a}^{(j)}$. All this is done within time *poly*(q), and we know that $\overline{a} \in q(D)$ if, and only if, all these tests succeed. This completes the proof of part (a) of Theorem 3.4.

Let us now turn to the proof of part (b) of Theorem 3.4. We are given a query $q := q_\varphi(z_1, \ldots, z_k)$, and without loss of generality we assume that q is a homomorphic core and q is not t-hierarchical. Thus, q violates condition (i) or (ii) of Definition 3.3. In case that it violates condition (i), the proof is virtually identical to the proof of Theorem 3.4 in [5] (see [6] for a proof). Let us consider the case where q violates condition (ii) of Definition 3.3. In this case, there are two variables $x \in \text{free}(q)$ and $y \in \text{vars}(q) \setminus \text{free}(q)$ and two atoms $\psi^{x,y}$ and ψ^y of q with $\text{vars}(\psi^{x,y}) \cap \{x, y\} = \{x, y\}$ and $\text{vars}(\psi^y) \cap \{x, y\} = \{y\}$. The easiest example of a query for which this is true is $q_{E\text{-}T} := \{ (x) : \exists y \, (Exy \wedge Ty) \}$. Here, we illustrate the proof idea for the particular query $q_{E\text{-}T}$; a proof for the general case can be found in [6].

Assume that there is a dynamic algorithm that solves the testing problem for $q_{E\text{-}T}$ with update time $t_u = O(n^{1-\epsilon})$ and testing time $t_t = O(n^{1-\epsilon})$ on databases whose active domain is of size $O(n)$. We show how this algorithm can be used to solve the OuMv-problem.

For the OuMv-problem, we receive as input an $n \times n$ matrix M. We start the preprocessing phase of our testing algorithm for $q_{E\text{-}T}$ with the empty database $D = (E^D, T^D)$ where $E^D = T^D = \emptyset$. As this database has constant size, the preprocessing is finished in constant time. We then apply $O(n^2)$ update steps to ensure that $E^D = \{(i, j) : M_{i,j} = 1\}$. All this takes time at most $O(n^2) \cdot t_u = O(n^{3-\epsilon})$. Throughout the remainder of the construction, we will never change E^D, and we will always ensure that $T^D \subseteq [n]$.

When we receive two vectors $\vec{u}^{\,t}$ and $\vec{v}^{\,t}$ in the dynamic phase of the OuMv-problem, we proceed as follows. First, we perform the update commands $\mathsf{delete}\,T(j)$ for each $j \in [n]$ with

$\vec{v}^{\,t}_j = 0$, and the update commands insert $T(j)$ for each $j \in [n]$ with $\vec{v}^{\,t}_j = 1$. This is done within time $n \cdot t_u = O(n^{2-\epsilon})$. By construction of D we know that for every $i \in [n]$ we have

$$i \;\in\; q_{E\text{-}T}(D) \quad\iff\quad \text{there is a } j \in [n] \text{ such that } M_{i,j} = 1 \text{ and } \vec{v}^{\,t}_j = 1\,.$$

Thus, $(\vec{u}^{\,t})^{\mathsf{T}} M \vec{v}^{\,t} = 1 \iff$ there is an $i \in [n]$ with $\vec{u}^{\,t}_i = 1$ and $i \in q_{E\text{-}T}(D)$. Therefore, after having called the **test** routine for $q_{E\text{-}T}$ for each $i \in [n]$ with $\vec{u}^{\,t}_i = 1$, we can output the correct result of $(\vec{u}^{\,t})^{\mathsf{T}} M \vec{v}^{\,t}$. This takes time at most $n \cdot t_t = O(n^{2-\epsilon})$. I.e., for each $t \in [n]$ after receiving the vectors $\vec{u}^{\,t}$ and $\vec{v}^{\,t}$, we can output $(\vec{u}^{\,t})^{\mathsf{T}} M \vec{v}^{\,t}$ within time $O(n^{2-\epsilon})$. Consequently, the overall running time for solving the OuMv-problem is bounded by $O(n^{3-\epsilon})$.

Using the technical machinery of [5], this can be generalised from $q_{E\text{-}T}$ to all queries q that violate condition (ii) of Definition 3.3. This completes the proof of Theorem 3.4. ◀

4 Unions of conjunctive queries

In this section we consider dynamic query evaluation for UCQs. To transfer our notions of *hierarchical* queries from CQs to UCQs, we say that a UCQ $q(\overline{u})$ of the form $q_1(\overline{u}_1) \cup \cdots \cup q_d(\overline{u}_d)$ is q-hierarchical (t-hierarchical) if every CQ $q_i(\overline{u}_i)$ in the union is q-hierarchical (t-hierarchical). Note that for *Boolean* queries (CQs as well as UCQs) the notions of being q-hierarchical and being t-hierarchical coincide, and for a k-ary UCQ q it can be checked in time *poly*(q) if q is q-hierarchical or t-hierarchical.

Testing. The following theorem generalises the statement of Theorem 3.4 from CQs to UCQs. Its proof follows easily from the Theorems 3.4 and 3.2; see [6] for details.

▶ **Theorem 4.1.**

(a) *There is a dynamic algorithm that receives a t-hierarchical k-ary UCQ q and a σ-db D_0, and computes within $t_p = poly(q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(q)$ and allows to test for an input tuple $\overline{a} \in \boldsymbol{dom}^k$ if $\overline{a} \in q(D)$ within time $t_t = poly(q)$. Furthermore, the algorithm allows to answer a t-hierarchical Boolean UCQ within time $t_a = O(1)$.*

(b) *Let $\epsilon > 0$ and let q be a k-ary UCQ that is not equivalent to a t-hierarchical UCQ. There is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\epsilon})$ update time that can test for any input tuple $\overline{a} \in \boldsymbol{dom}^k$ if $\overline{a} \in q(D)$ within testing time $t_t = O(n^{1-\epsilon})$, unless the OMv-conjecture fails. Furthermore, if $k = 0$ (i.e., q is a Boolean UCQ), then there is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that answers $q(D)$ in time $t_a = O(n^{2-\varepsilon})$, unless the OMv-conjecture fails.*

Enumerating. It turns out that q-hierarchical UCQs, like q-hierarchical CQs, allow for efficient enumeration under updates. This, and the according lower bound, is stated in the following Theorem 4.2. In contrast to Theorem 4.1, the result does not follow immediately from the tractability of the enumeration problem for q-hierarchical CQs, because one has to ensure that tuples from result sets of two different CQs are not reported twice while enumerating their union.

▶ **Theorem 4.2.**

(a) *There is a dynamic algorithm that receives a q-hierarchical k-ary UCQ q and a σ-db D_0, and computes within $t_p = poly(q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(q)$ and allows to enumerate $q(D)$ with delay $t_d = poly(q)$.*

(b) *Let $\epsilon > 0$ and let q be a k-ary UCQ whose homomorphic core is not q-hierarchical and is a union of self-join-free CQs. There is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that enumerates $q(D)$ with delay $t_d = O(n^{1-\varepsilon})$, unless the OMv-conjecture fails.*

To prove Theorem 4.2 (a), we first develop a general method for enumerating the union of sets. We say that a data structure for a set T *allows to skip* if it is possible to test whether $t \in T$ in constant time and for some ordering $t_1, \dots, t_n$ of the elements in T there are a function start, which returns t_1 in constant time, and a function $\mathsf{next}(t_i)$, which returns t_{i+1} (if $i < n$) or $\mathtt{EOE}$ (if $i = n$) in constant time. Note that a data structure that allows to skip enables constant delay enumeration of t_i, t_{i+1}, …, t_n starting from an arbitrary element $t_i \in T$ (but we do not have control over the underlying order). An example of such a data structure is an explicit representation of the elements of T in a linked list with constant access. Another example is the data structure of the enumeration algorithm for the result $T := q(D)$ of a q-hierarchical CQ q, provided by Theorem 3.2 (aii)&(aiv). The next lemma states that we can use these data structures for sets T_j to enumerate the union $\bigcup_j T_j$ with constant delay and without repetition.

▶ **Lemma 4.3.** *Let $\ell \geqslant 1$ and let $T_1, \dots, T_\ell$ be sets such that for each $j \in [\ell]$ there is a data structure for T_j that allows to skip. Then there is an algorithm that enumerates, without repetition, all elements in $T_1 \cup \dots \cup T_\ell$ with $O(\ell)$ delay.*

Proof. For each $i \in [\ell]$ let start^i and next^i be the start element and the iterator for the set T_i. The main idea for enumerating the union $T_1 \cup \dots \cup T_\ell$ is to first enumerate all elements in T_1, and then $T_2 \setminus T_1$, $T_3 \setminus (T_1 \cup T_2)$, …, $T_\ell \setminus (T_1 \cup \dots \cup T_{\ell-1})$. In order to do this we have to exclude all elements that have already been reported from all subsequent sets. As we want to ensure constant delay enumeration, we cannot just ignore the elements in $T_i \cap (T_1 \cup \dots \cup T_{i-1})$ while enumerating T_i. As a remedy, we use an additional pointer to jump from an element that has already been reported to the least element that needs to be reported next. To do this we use arrays skip^i (for all $i \in [\ell]$) to jump over excluded elements. For technical reasons we add for each set T_i a dummy element $\mathtt{EOE}$ at the end of its list representation. The algorithm preserves the following invariant: "If $t_r, \dots, t_s$ is a maximal interval of elements in T_i that have already been reported, then $\mathsf{skip}^i[t_r] = t_{s+1}$." For technical reasons we also need the array $\mathsf{skipback}^i$ which represents the inverse pointer, i.e., $\mathsf{skipback}^i[t_{s+1}] = t_r$. It follows from the invariant that $\mathsf{skip}^i[t] \neq \mathtt{nil}$ implies $\mathsf{skip}^i[\mathsf{skip}^i[t]] = \mathtt{nil}$. As a consequence, Algorithm 1 enumerates elements with constant delay. It uses the procedure $\textsc{exclude}^j$ described in Algorithm 2 to update the arrays whenever an element t has been reported (note that every element is excluded at most once). See Figure 1 in the appendix for an illustration. It is straightforward to verify that these algorithms provide the desired functionality within the claimed time bounds. ◀

Proof of Theorem 4.2. The upper bound follows immediately from combining Lemma 4.3 with Theorem 3.2 (aiv). For the lower bound let q_i be a self-join-free non-q-hierarchical CQ in the homomorphic core q' of the UCQ q. For every database D that maps homomorphically into q_i it holds that $q_j(D) = \emptyset$ for every other CQ q_j in q' (with $j \neq i$), since otherwise there would be a homomorphism from q_j to D and hence to q_i, contradicting that q' is a homomorphic core. It follows that every dynamic algorithm that enumerates the result of q on a database D which maps homomorphically into q_i also enumerates $q_i(D) = q(D)$, contradicting Theorem 3.2 (biii). ◀

Algorithm 1 Enumeration algorithm for $T_1 \cup \cdots \cup T_\ell$.

```
Input: Data structures for sets T_j with first element start^j and iterator next^j.
Pointer skip^j[t] = skipback^j[t] = nil for all j ∈ [ℓ] and t ∈ T_j.

for i = 1, ..., ℓ do
    t = start^i
    while t ≠ EOE do
        if skip^i[t] == nil then
            Output element t
            for j = i + 1 → ℓ do
                EXCLUDE^j(t)
            t = next^i(t)
        else
            t = skip^i[t]
Output the end-of-enumeration message EOE.
```

Algorithm 2 Procedure $\textsc{exclude}^j$ for excluding t from T_j.

```
if t ∈ T_j then
    if skipback^j[t] ≠ nil then
        t⁻ = skipback^j[t]
        skipback^j[t] = nil
    else
        t⁻ = t
    if skip^j[next^j(t)] ≠ nil then
        t⁺ = skip^j[next^j(t)]
        skip^j[next^j(t)] = nil
    else
        t⁺ = next^j(t)
    skip^j[t⁻] = t⁺;   skipback^j[t⁺] = t⁻
```

Counting. Note that according to Theorem 3.2, for CQs the enumeration problem as well as the counting problem can be solved by efficient dynamic algorithms if, and (modulo algorithmic conjectures) only if, the query is q-hierarchical. In contrast to this, it turns out that for UCQs computing the number of output tuples can be much harder than enumerating the query result. To characterise the UCQs that allow for efficient dynamic counting algorithms, we use the following notation. For two k-ary CQs $q_\varphi(u_1, \ldots, u_k)$ and $q_\psi(v_1, \ldots, v_k)$ we define the intersection $q := q_\varphi \cap q_\psi$ to be the following k-ary query. If there is an $i \in [k]$ such that u_i and v_i are distinct elements from **dom**, then $q := \emptyset$ (and this query is q-hierarchical by definition). Otherwise, we let $w_1, \ldots, w_k$ be elements from $\mathbf{var} \cup \mathbf{dom}$ which satisfy the following for all $i, j \in [k]$ and all $a \in \mathbf{dom}$:

$$\big(\ w_i = a \iff u_i = a \text{ or } v_i = a\ \big) \quad \text{and} \quad \big(\ w_i = w_j \iff u_i = u_j \text{ or } v_i = v_j\ \big).$$

We obtain φ' from φ (and ψ' from ψ) by replacing every $u_i \in \{u_1, \ldots, u_k\} \cap \text{free}(\varphi)$ (and $v_i \in \{v_1, \ldots, v_k\} \cap \text{free}(\psi)$) by w_i. Finally, we let $q = \{\ (w_1, \ldots, w_k)\ :\ \varphi' \wedge \psi'\ \}$, where we can assume that $\varphi' \wedge \psi'$ is (equivalent to) a conjunctive formula of the form $(*)$. Note that for every database D it holds that $q(D) = q_\varphi(D) \cap q_\psi(D)$.

To compute the number of result tuples in a UCQ $q = \bigcup_{i \in [d]} q_i(\overline{u}_i)$ we first define for every $I \subseteq [d]$ the CQ $q_I = \bigcap_{i \in I} q_i$. To take care of equivalent queries q_I and $q_{I'}$ we define the equivalence relation $I \cong I' \iff q_I \equiv q_{I'}$ and let $\mathfrak{P}$ be the partition of $\{I : \emptyset \neq I \subseteq [d]\}$ into equivalence classes. For an $\mathcal{I} \in \mathfrak{P}$ we denote by $q_{\mathcal{I}}$ the common homomorphic core of all q_I, $I \in \mathcal{I}$, and define $a_{\mathcal{I}} := \sum_{I \in \mathcal{I}} (-1)^{|I|+1}$. By the inclusion-exclusion principle we get:

$$|q(D)| \quad = \sum_{\emptyset \neq I \subseteq [d]} (-1)^{|I|+1} \cdot |q_I(D)| \quad = \quad \sum_{\mathcal{I} \in \mathfrak{P}} a_{\mathcal{I}} \cdot |q_{\mathcal{I}}(D)|. \tag{1}$$

If all $q_{\mathcal{I}}$ with non-zero coefficients $a_{\mathcal{I}}$ are q-hierarchical, then we can compute the result size of the UCQ as a linear combination of a constant number of q-hierarchical CQs. We will show in Theorem 4.5 that this approach is indeed optimal, justifying the following definition.

▶ **Definition 4.4.** A UCQ q is *exhaustively q-hierarchical* if $q_{\mathcal{I}}$ is q-hierarchical for every $\mathcal{I} \in \mathfrak{P}$ with $a_{\mathcal{I}} \neq 0$.

Being exhaustively q-hierarchical is a stronger requirement than being q-hierarchical, e.g., the UCQ $\{(x,y) : Sx \wedge Exy\} \cup \{(x,y) : Exy \wedge Ty\}$ is q-hierarchical, but not exhaustively q-hierarchical. The straightforward way of deciding whether a UCQ q is exhaustively q-hierarchical requires time $2^{poly(q)}$, and it is open whether this can be improved. The next theorem shows that the exhaustively q-hierarchical queries are precisely those UCQs that allow for efficient dynamic counting algorithms.

▶ **Theorem 4.5.**

(a) *There is a dynamic algorithm that receives an exhaustively q-hierarchical UCQ q and a σ-db D_0, computes in $t_p = 2^{poly(q)} \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = 2^{poly(q)}$ and computes $|q(D)|$ in time $t_c = O(1)$.*

(b) *Let $\epsilon > 0$ and let q be a UCQ that is not exhaustively q-hierarchical. There is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that computes $|q(D)|$ in time $t_c = O(n^{1-\varepsilon})$, unless the OMv-conjecture or the OV-conjecture fails.*

Proof. Part (a) follows from the upper bound of Theorem 3.2 (ai) and the inclusion-exclusion argument (1). For proving part (b) let $\mathfrak{H} \subseteq \mathfrak{P}$ be the set of equivalence classes $\mathcal{I}$ such that $a_{\mathcal{I}} \neq 0$ and $q_{\mathcal{I}}$ is q-hierarchical, and let $\mathfrak{N} \subseteq \mathfrak{P}$ be the set of equivalence classes $\mathcal{I}$ such that $a_{\mathcal{I}} \neq 0$ and $q_{\mathcal{I}}$ is *not* q-hierarchical. By Definition 4.4 we have that $\mathfrak{N} \neq \emptyset$. Moreover, since for all distinct $\mathcal{I}, \mathcal{I}' \in \mathfrak{N}$ the queries $q_{\mathcal{I}}$ and $q_{\mathcal{I}'}$ are not homomorphically equivalent, we can choose a $\mathcal{J} \in \mathfrak{N}$, which is *minimal* in the sense that for every $\mathcal{I} \in \mathfrak{N} \setminus \{\mathcal{J}\}$ there is no homomorphism from $q_{\mathcal{I}}$ to $q_{\mathcal{J}}$.

Now suppose that D is a database from the class of databases that map homomorphically into $q_{\mathcal{J}}$ and let $h\colon D \to q_{\mathcal{J}}$ be a homomorphism. For every $\mathcal{I} \in \mathfrak{N} \setminus \{\mathcal{J}\}$ it holds that there is no homomorphism $h'\colon q_{\mathcal{I}} \to D$, since otherwise $h \circ h'$ would be a homomorphism from $q_{\mathcal{I}}$ to $q_{\mathcal{J}}$. Hence, $q_{\mathcal{I}}(D) = \emptyset$ for all $\mathcal{I} \in \mathfrak{N} \setminus \{\mathcal{J}\}$ and thus, by (1) we have

$$|q(D)| \quad = \quad a_{\mathcal{J}} \cdot |q_{\mathcal{J}}(D)| \quad + \sum_{\mathcal{I} \in \mathfrak{H}} a_{\mathcal{I}} \cdot |q_{\mathcal{I}}(D)|.$$

Assume for contradiction that we can efficiently compute $|q(D)|$ with update time t_u and counting time t_c. By Theorem 3.2 (ai) we can maintain $|q_{\mathcal{I}}(D)|$ for each $\mathcal{I} \in \mathfrak{H}$ with update time $poly(q_{\mathcal{I}})$ and counting time $O(1)$. Thus, we can compute $|q_{\mathcal{J}}(D)| = \frac{1}{a_{\mathcal{J}}} \big(|q(D)| - \sum_{\mathcal{I} \in \mathfrak{H}} a_{\mathcal{I}} \cdot |q_{\mathcal{I}}(D)| \big)$ with update time $t_u + 2^{poly(q)}$ and counting time $t_c + 2^{poly(q)}$. Since $q_{\mathcal{J}}$ is a non-q-hierarchical homomorphic core, the lower bound for maintaining $|q(D)|$ follows from Theorem 3.2 (bii). This completes the proof of Theorem 4.5. ◀

5 CQs and UCQs with integrity constraints

In the presence of integrity constraints, the characterisation of tractable queries changes and depends on the query as well as on the set of constraints. When considering a scenario where databases are required to satisfy a set Σ of constraints, we allow to execute a given **update** command only if the resulting database still satisfies all constraints in Σ. When speaking of *(σ, Σ)-dbs* we mean σ-dbs D that satisfy all constraints in Σ. Two queries q and q' are *Σ-equivalent* (for short: $q \equiv_\Sigma q'$) if $q(D) = q'(D)$ for every (σ, Σ)-db D.

We first consider *small domain constraints*, i.e., constraints δ of the form $R[i] \subseteq C$ where $R \in \sigma$, $i \in \{1, \ldots, \text{ar}(R)\}$, and $C \subseteq \textbf{dom}$ is a finite set. A σ-db D *satisfies* δ if $\pi_i(R^D) \subseteq C$.

For these constraints we are able to give a clear picture of the tractability landscape by reducing CQs and UCQs with small domain constraints to UCQs without integrity constraints and applying the characterisations for UCQs achieved in Section 4. We start with an example that illustrates how a query can be simplified in the presence of small domain constraints.

▶ **Example 5.1.** Consider the Boolean query $q_{S\text{-}E\text{-}T} := \{\, () \,:\, \exists x \exists y\, (\, Sx \wedge Exy \wedge Ty \,)\, \}$, which is not q-hierarchical. By Theorem 3.2 it cannot be answered by a dynamic algorithm with sublinear update time and sublinear answer time, unless the OMv-conjecture fails. But in the presence of the small domain constraint $\delta_{sd} := S[1] \subseteq C$ for a set $C = \{a_1, \ldots, a_c\} \subseteq \textbf{dom}$, the query $q_{S\text{-}E\text{-}T}$ is $\{\delta_{sd}\}$-equivalent to the q-hierarchical UCQ $q' := \bigcup_{a_i \in C} \{\, () \,:\, \exists y\, (\, Sa_i \wedge Ea_iy \wedge Ty \,)\, \}$. Therefore, by Theorem 4.1, q' and hence $q_{S\text{-}E\text{-}T}$ can be answered with constant update time and constant answer time on all databases that satisfy δ_{sd}.

For handling the general case, assume we are given a set Σ of small domain constraints and an arbitrary k-ary CQ q of the form $(**)$ where φ is of the form $(*)$. We define a function $Dom_{q,\Sigma}$ that maps each $x \in \text{vars}(q)$ to a set $Dom_{q,\Sigma}(x) \subseteq \textbf{dom}$ as follows. As an initialisation let $f(x) = \textbf{dom}$ for each $x \in \text{vars}(q)$. Consider each constraint δ in Σ and let $S[i] \subseteq C$ be the form of δ. Consider each atom ψ_j of φ and let $Rv_1 \cdots v_r$ be the form of ψ_j. If $R = S$ and $v_i \in \textbf{var}$, then let $f(v_i) := f(v_i) \cap C$. We let $Dom_{q,\Sigma}$ be the mapping f obtained at the end of this process and define the set of variables of q that are restricted by Σ by $rvars_\Sigma(q) := \{x \in \text{vars}(q) \,:\, Dom_{q,\Sigma}(x) \neq \textbf{dom}\}$. Let $M_{q,\Sigma}$ be the set of all mappings $\alpha : V \to \textbf{dom}$ with $V = rvars_\Sigma(q)$ and $\alpha(x) \in Dom_{q,\Sigma}(x)$ for each $x \in V$. Note that $M_{q,\Sigma}$ is finite; and it is empty if, and only if, $Dom_{q,\Sigma}(x) = \emptyset$ for some $x \in \text{vars}(q)$. For a mapping $\alpha : V \to \textbf{dom}$ with $V \subseteq \textbf{var}$ we let q_α be the k-ary CQ obtained from q as follows: for each $x \in V$, if present in q, the existential quantifier "$\exists x$" is omitted, and afterwards every occurrence of x in q is replaced with the constant $\alpha(x)$. Clearly, $q_\alpha(D) \subseteq q(D)$ for every σ-db D. With these notations, we obtain the following (the proof can be found in [6]).

▶ **Lemma 5.2.** *For a CQ q and a set Σ of small domain constraints, let $M := M_{q,\Sigma}$. If $M = \emptyset$, then $q(D) = \emptyset$ for every (σ, Σ)-db D. Otherwise, q is Σ-equivalent to the UCQ $q_\Sigma := \bigcup_{\alpha \in M} q_\alpha$.*

This reduction from a CQ q to a UCQ q_Σ directly translates to UCQs: if q is a union of the CQs $q_1, \ldots, q_d$, then we let $q_\Sigma := \bigcup_{i \in [d]} (q_i)_\Sigma$. Note that if the UCQ q is a homomorphic core, then so is q_Σ. Therefore, the following dichotomy theorem for UCQs under small domain constraints is a direct consequence of Lemma 5.2 and the Theorems 4.1, 4.2, and 4.5.

▶ **Theorem 5.3.** *Let q be a UCQ that is a homomorphic core and Σ a set of small domain constraints with $M_{q,\Sigma} \neq \emptyset$. Suppose that the OMv-conjecture and the OV-conjecture hold.*

(1a) *If q_Σ is t-hierarchical, then q can be tested on (σ, Σ)-dbs in constant time with linear preprocessing time and constant update time.*

(1b) *If q_Σ is not t-hierarchical, then on the class of (σ,Σ)-dbs testing in time $O(n^{1-\epsilon})$ is not possible with $O(n^{1-\epsilon})$ update time.*

(2a) *If q_Σ is q-hierarchical, then there is a data structure with linear preprocessing and constant update time that allows to enumerate $q(D)$ with constant delay on (σ,Σ)-dbs.*

(2b) *If q_Σ is not q-hierarchical and in addition self-join-free, then $q(D)$ cannot be enumerated with $O(n^{1-\epsilon})$ delay and $O(n^{1-\epsilon})$ update time on (σ,Σ)-dbs.*

(3a) *If q_Σ is exhaustively q-hierarchical, then there is data structure with linear preprocessing and constant update time that allows to compute $|q(D)|$ in constant time on (σ,Σ)-dbs.*

(3b) *If q_Σ is not exhaustively q-hierarchical, then computing $|q(D)|$ on (σ,Σ)-dbs in time $O(n^{1-\epsilon})$ is not possible with $O(n^{1-\epsilon})$ update time.*

Thus, the tractability of a UCQ q on (σ,Σ)-dbs only depends on the structure of the query q_Σ. Note that while the size of q_Σ might be $c^{O(q)}$, where c is the largest number of constants in a small domain, it can be checked in time *poly*(q) whether q_Σ is (t- or q-)hierarchical.

Let us take a brief look at two other kinds of constraints: inclusion dependencies and functional dependencies, which both can also cause a hard query to become tractable. An *inclusion dependency* δ is of the form $R[i_1,\dots,i_m] \subseteq S[j_1,\dots,j_m]$ where $R,S \in \sigma$, $m \geqslant 1$, $i_1,\dots,i_m \in \{1,\dots,\mathrm{ar}(R)\}$, and $j_1,\dots,j_m \in \{1,\dots,\mathrm{ar}(S)\}$. A σ-db D *satisfies* δ if $\pi_{i_1,\dots,i_m}(R^D) \subseteq \pi_{j_1,\dots,j_m}(S^D)$. As an example, consider the query $q_{\textit{S-E-T}}$ and the inclusion dependency $\delta_{\textit{ind}} := E[2] \subseteq T[1]$. Obviously, $q_{\textit{S-E-T}}$ is $\{\delta_{\textit{ind}}\}$-equivalent to the q-hierarchical (and hence easy) CQ $q' := \{\, () \;:\; \exists x \exists y\ (\ Sx \wedge Exy\ )\,\}$. To turn this into a general principle, we say that an inclusion dependency δ of the form $R[i_1,\dots,i_m] \subseteq S[j_1,\dots,j_m]$ *can be applied* to a CQ q if q contains an atom ψ_1 of the form $Rv_1\cdots v_r$ and an atom ψ_2 of the form $Sw_1\cdots w_s$ such that

1. $(v_{i_1},\dots,v_{i_m}) = (w_{j_1},\dots,w_{j_m})$,
2. for all $j \in [s] \setminus \{j_1,\dots,j_m\}$ we have $w_j \in \mathbf{var}$, $w_j \notin \mathrm{free}(q)$, $\mathrm{atoms}(w_j) = \{\psi_2\}$, and
3. for all $j,j' \in [s] \setminus \{j_1,\dots,j_m\}$ with $j \neq j'$ we have $w_j \neq w_{j'}$;

and *applying δ to q at (ψ_1,ψ_2)* then yields the CQ q' which is obtained from q by omitting the atom ψ_2 and omitting the quantifiers $\exists z$ for all $z \in \mathrm{vars}(\psi_2) \setminus \{w_{j_1},\dots,w_{j_m}\}$. By this construction we have $\mathrm{vars}(q') = \mathrm{vars}(q) \setminus \{w_j \;:\; j \in [s] \setminus \{j_1,\dots,j_m\}\}$.

▶ **Claim 5.4.** *$q' \equiv_{\{\delta\}} q$, and if q is q-hierarchical, then so is q'.*

See [6] for a proof. From the claim it follows that we can simplify a query by iteratively applying inclusion dependencies to pairs of atoms of the query. In some cases, this transforms queries that are hard in general into Σ-equivalent queries that are q-hierarchical and hence easy for dynamic evaluation. E.g., an iterated application of $\delta_{\textit{ind}} := E[2] \subseteq E[1]$ transforms the non-t-hierarchical query $\{\,(x,y) \;:\; \exists z_1 \exists z_2\, (\, Exy \wedge Eyz_1 \wedge Ez_1z_2\,)\,\}$ into the q-hierarchical query $\{\,(x,y) \;:\; Exy\,\}$. However, the limitations of this approach are documented by the query $q := \{\,() \;:\; \exists x \exists y \exists z \exists z'\,(\, Sx \wedge Exy \wedge Ty \wedge Rzz'\,)\,\}$, which is Σ-equivalent to the q-hierarchical query $q' := \{\,() \;:\; \exists z \exists z'\ Rzz'\,\}$, for $\Sigma := \{\, R[1,2] \subseteq E[1,2]\ ,\ R[1] \subseteq S[1]\ ,\ R[2] \subseteq T[1]\,\}$, but where q' cannot be obtained by iteratively applying dependencies of Σ to q.

Also, the presence of *functional dependencies* can cause a hard query to become tractable: Consider the functional dependency $\delta_{\textit{fd}} := E[1 \to 2]$, which is satisfied by a database D iff for every $a \in \mathbf{dom}$ there is at most one $b \in \mathbf{dom}$ such that $(a,b) \in E^D$. On databases that satisfy $\delta_{\textit{fd}}$, the query $q_{\textit{S-E-T}}$ can be evaluated with constant answer time and constant update time as follows: One can store for every b the number m_b of elements $(a,b) \in E^D$ such that $a \in S^D$, and, in addition, the number $m = \sum_{b \in T^D} m_b$, which is non-zero if and only if $q_{\textit{S-E-T}}(D) = \texttt{yes}$. The functional dependency guarantees that every update affects at most one number m_b and one summand of m. Using constant access data structures, the query result can therefore be maintained with constant update time.

The nature of this example is somewhat different compared to the approaches for small domain constraints or inclusion constraints described above: We can show that the query becomes tractable, but we are not aware of any $\{\delta_{fd}\}$-equivalent q-hierarchical CQ or UCQ that would explain its tractability via a reduction to the setting without integrity constraints. To exploit the full power of functional dependencies for improving dynamic query evaluation, it seems therefore necessary to come up with new algorithmic approaches that go beyond the techniques we have for (q- or t-)hierarchical queries.

References

1 Amir Abboud, Richard Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 218–230. SIAM, 2015. doi:10.1137/1.9781611973730.17.

2 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: http://webdam.inria.fr/Alice/.

3 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007. doi:10.1007/978-3-540-74915-8_18.

4 Pablo Barceló, Georg Gottlob, and Andreas Pieris. Semantic acyclicity under constraints. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 343–354. ACM, 2016. doi:10.1145/2902251.2902302.

5 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 303–318. ACM, 2017. doi:10.1145/3034786.3034789.

6 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering ucqs under updates and in the presence of integrity constraints. *CoRR*, abs/1709.10039, 2017. arXiv:1709.10039.

7 Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 77–90. ACM, 1977. doi:10.1145/800105.803397.

8 Hubie Chen and Stefan Mengel. A trichotomy in the complexity of counting answers to conjunctive queries. In Marcelo Arenas and Martín Ugarte, editors, *18th International Conference on Database Theory, ICDT 2015, March 23-27, 2015, Brussels, Belgium*, volume 31 of *LIPIcs*, pages 110–126. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.ICDT.2015.110.

9 Hubie Chen and Stefan Mengel. Counting answers to existential positive queries: A complexity classification. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 315–326. ACM, 2016. doi:10.1145/2902251.2902279.

10 Rada Chirkova and Jun Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012. doi:10.1561/1900000020.

11 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

12 Víctor Dalmau and Peter Jonsson. The complexity of counting homomorphisms seen from the other side. *Theor. Comput. Sci.*, 329(1-3):315–323, 2004. doi:10.1016/j.tcs.2004.08.008.

13 Arnaud Durand and Stefan Mengel. Structural tractability of counting of solutions to conjunctive queries. *Theory Comput. Syst.*, 57(4):1202–1249, 2015. doi:10.1007/s00224-014-9543-y.

14 Georg Gottlob, Stephanie Tien Lee, Gregory Valiant, and Paul Valiant. Size and treewidth bounds for conjunctive queries. *J. ACM*, 59(3):16:1–16:35, 2012. doi:10.1145/2220357.2220363.

15 Gianluigi Greco and Francesco Scarcello. Counting solutions to conjunctive queries: structural and hybrid tractability. In Richard Hull and Martin Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 132–143. ACM, 2014. doi:10.1145/2594538.2594559.

16 Martin Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *J. ACM*, 54(1):1:1–1:24, 2007. doi:10.1145/1206035.1206036.

17 Martin Grohe, Thomas Schwentick, and Luc Segoufin. When is the evaluation of conjunctive queries tractable? In Jeffrey Scott Vitter, Paul G. Spirakis, and Mihalis Yannakakis, editors, *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 657–666. ACM, 2001. doi:10.1145/380752.380867.

18 Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD'93, Washington, D.C., USA, May 25–28, 1993*, pages 157–166. ACM, 1993. doi:10.1145/170036.170066.

19 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 21–30. ACM, 2015. doi:10.1145/2746539.2746609.

20 Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1259–1274. ACM, 2017. doi:10.1145/3035918.3064027.

21 Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. Computing join queries with functional dependencies. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 327–342. ACM, 2016. doi:10.1145/2902251.2902289.

22 Christoph Koch. Incremental query evaluation in a ring of databases. In Jan Paredaens and Dirk Van Gucht, editors, *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 87–98. ACM, 2010. doi:10.1145/1807085.1807100.

23 Christoph Koch, Daniel Lupei, and Val Tannen. Incremental view maintenance for collection programming. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 75–90. ACM, 2016. doi:10.1145/2902251.2902286.

24 Dániel Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *J. ACM*, 60(6):42:1–42:51, 2013. doi:10.1145/2535926.

25 Bernard M. E. Moret and Henry D. Shapiro. *Algorithms from P to NP: Volume 1: Design & Efficiency*. Benjamin-Cummings, 1991.

26 Milos Nikolic, Mohammad Dashti, and Christoph Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 511–526. ACM, 2016. doi:10.1145/2882903.2915246.

27 Sushant Patnaik and Neil Immerman. Dyn-fo: A parallel, dynamic complexity class. *J. Comput. Syst. Sci.*, 55(2):199–209, 1997. doi:10.1006/jcss.1997.1520.

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

29 Thomas Zeume and Thomas Schwentick. Dynamic conjunctive queries. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.*, pages 38–49. OpenProceedings.org, 2014. doi:10.5441/002/icdt.2014.08.

A Proof of Lemma 3.5

Proof. Along Definition 3.3 it is straightforward to construct an algorithm which decides in time *poly*(q) whether a given CQ q is t-hierarchical.

Let $q := q_\varphi(z_1, \ldots, z_k)$ be a given t-hierarchical CQ. Let A_0 be the set of all atoms ψ of q with $\text{vars}(\psi) \subseteq \text{free}(q)$, and let φ_0 be the quantifier-free conjunctive formula $\varphi_0 := \bigwedge_{\psi \in A_0} \psi$. For each $Z \subseteq \text{free}(q)$ let A_Z be the set of all atoms ψ of q such that $Z = \text{vars}(\psi) \cap \text{free}(q)$ and $\text{vars}(\psi) \supsetneq Z$. Let $Z_1, \ldots, Z_n$ (for $n \geqslant 0$) be a list of all those $Z \subseteq \text{free}(q)$ with $A_Z \neq \emptyset$. For each $j \in [n]$ let $A_j := A_{Z_j}$ and let $Y_j := \big(\bigcup_{\psi \in A_j} \text{vars}(\psi)\big) \setminus Z_j$.

▶ **Claim A.1.** *$Y_j \cap Y_{j'} = \emptyset$ for all $j, j' \in [n]$ with $j \neq j'$.*

Proof. We know that $Z_j \neq Z_{j'}$. W.l.o.g. there is a $z \in Z_j$ with $z \notin Z_{j'}$.

For contradiction, assume that $Y_j \cap Y_{j'}$ contains some variable y. Then, $y \in \text{vars}(\psi)$ for some $\psi \in A_j$ and $y \in \text{vars}(\psi')$ for some $\psi' \in A_{j'}$. By definition of A_j we know that $\text{vars}(\psi) \cap \text{free}(q) = Z_j$, and hence $z \in \text{vars}(\psi)$. By definition of $A_{j'}$ we know that $\text{vars}(\psi') \cap \text{free}(q) = Z_{j'}$, and hence $z \notin \text{vars}(\psi')$. Hence, $\psi \in \text{atoms}(z)$ and $\psi' \notin \text{atoms}(z)$. Since $\psi \in \text{atoms}(y)$ and $\psi' \in \text{atoms}(y)$, we obtain that $\text{atoms}(z) \cap \text{atoms}(y) \neq \emptyset$ and $\text{atoms}(y) \not\subseteq \text{atoms}(z)$. But by assumption, q is t-hierarchical, and this contradicts condition (ii) of Definition 3.3. ◀

For each $j \in [n]$ consider the conjunctive formula $\varphi_j := \exists y_1^{(j)} \cdots \exists y_{\ell_j}^{(j)} \bigwedge_{\psi \in A_j} \psi$, where $\ell_j := |Y_j|$ and $(y_1^{(j)}, \ldots, y_{\ell_j}^{(j)})$ is a list of all variables in Y_j. Using Claim A.1, it is straightforward to see that $q' := \{\, (z_1, \ldots, z_k) \,:\, \varphi_0 \wedge \bigwedge_{j \in [n]} \varphi_j \,\}$ is a generalised CQ that is equivalent to q. Furthermore, q' can be constructed in time *poly*(q). To complete the proof of Lemma 3.5 we consider for each $j \in [n]$ the CQ $q_j := \{\, \overline{z}^{(j)} \,:\, \varphi_j \,\}$, where $\overline{z}^{(j)}$ is a tuple of length $|Z_j|$ consisting of all the variables in Z_j.

▶ **Claim A.2.** *q_j is q-hierarchical, for each $j \in [n]$.*

Proof. First of all, note that q_j satisfies condition (ii) of Definition 3.1, since $\mathrm{free}(q_j) = Z_j$, $\mathrm{atoms}_{q_j}(z) = A_j$ for every $z \in Z_j$, and $\mathrm{atoms}_{q_j}(y) \subseteq A_j$ for every $y \in Y_j = \mathrm{vars}(q_j) \setminus \mathrm{free}(q_j)$.

For contradiction, assume that q_j is not q-hierarchical. Then, q_j violates condition (i) of Definition 3.1. I.e., there are variables $x, x' \in Z_j \cup Y_j$ and atoms $\psi_1, \psi_2, \psi_3 \in A_j$ such that $\mathrm{vars}(\psi_1) \cap \{x, x'\} = \{x\}$, $\mathrm{vars}(\psi_2) \cap \{x, x'\} = \{x'\}$, and $\mathrm{vars}(\psi_3) \cap \{x, x'\} = \{x, x'\}$. Since $\mathrm{vars}(\psi) \cap \mathrm{free}(q) = Z_j$ for all $\psi \in A_j$, we know that $x, x' \notin \mathrm{free}(q)$. Therefore, $x, x' \in \mathrm{vars}(q) \setminus \mathrm{free}(q)$, and hence ψ_1, ψ_2, ψ_3 are atoms of q which witness that condition (i) of Definition 3.3 is violated. This contradicts the assumption that q is t-hierarchical. ◀

This completes the proof of Lemma 3.5. ◀

B Illustration of the effect of the EXCLUDE^j-Procedure used in the proof of Lemma 4.3

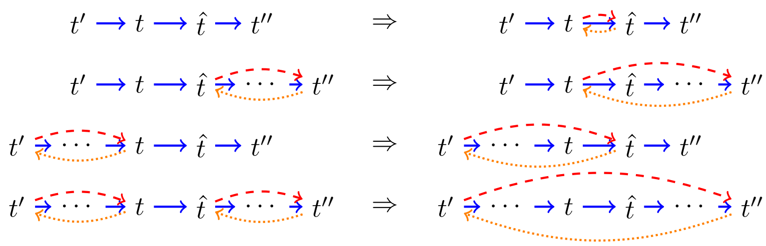

Figure 1 Modifications of the data structure caused by EXCLUDE$^j(t)$. In this illustration, a blue arrow from t to $\hat{t}$ means that $\hat{t} = \mathsf{next}^j(t)$; a dashed red arrow from t to $\hat{t}$ means that $\hat{t} = \mathsf{skip}^j[t]$, and a dotted red arrow from $\hat{t}$ to t means that $t = \mathsf{skipback}^j[\hat{t}]$.