**Noname manuscript No.**
(will be inserted by the editor)

# General Dynamic Yannakakis: Conjunctive Queries with Theta Joins Under Updates

**Muhammad Idris · Martín Ugarte · Stijn Vansummeren · Hannes Voigt · Wolfgang Lehner**

Received: date / Accepted: date

**Abstract** The ability to efficiently analyze changing data is a key requirement of many real-time analytics applications. In prior work, we have proposed General Dynamic Yannakakis (GDyn), a general framework for dynamically processing acyclic conjunctive queries with $\theta$-joins in the presence of data updates. Whereas traditional approaches face a trade-off between materialization of subresults (to avoid inefficient recomputation) and recomputation of subresults (to avoid the potentially large space overhead of materialization), GDyn is able to avoid this trade-off. It intelligently maintains a succinct data structure that supports efficient maintenance under updates and from which the full query result can quickly be enumerated. In this paper, we consolidate and extend the development of GDyn. First, we give full formal proof of GDyn's correctness and complexity. Second, we present a novel algorithm for computing GDyn query plans. Finally, we instantiate GDyn to the case where all $\theta$-joins are inequalities, and present extended experimental comparison against state of the art engines. Our approach performs consistently better than the competitor systems with multiple orders of magnitude improvements in both time and memory consumption.

M. Idris
Université Libre de Bruxelles, Belgium and TU Dresden, Germany
E-mail: midris@ulb.ac.be

M. Ugarte
Pontifical Catholic University of Chile, Chile
This work was done while the author was affiliated to ULB, Belgium
E-mail: martinugarte@uc.cl

S. Vansummeren
Université Libre de Bruxelles, Belgium
E-mail: svsummer@ulb.ac.be

Hannes Voigt
neo4j, Germany
This work was done while the author was affiliated to TU Dresden, Germany
E-mail: hannes.voigt@neo4j.com

Wolfgang Lehner
TU Dresden, Germany
E-mail: wolfgang.lehner@tu-dresden.de

## 1 Introduction

The ability to efficiently analyze changing data is a key requirement in Stream Processing [39], Complex Event Recognition [16], Business Intelligence [35], and Machine Learning [44]. Generally, the analysis that needs to be kept up-to-date, or at least their basic elements, are specified in a query language. The main task is then to efficiently update the query results under data updates.

In this context, we tackle the problem of *dynamic query evaluation*, where a given query $Q$ has to be evaluated against a database that is constantly changing. Concretely, when database $db$ is updated to database $db + u$ under update $u$, the objective is to efficiently compute $Q(db + u)$, taking into consideration that $Q(db)$ was already evaluated and re-computations could be avoided. Dynamic query evaluation is of utmost importance if response time requirements for queries under concurrent data updates have to be met or if data volumes are so large that full re-evaluation is prohibitive.

In this paper, we focus on the problem of dynamic evaluation for conjunctive queries that feature multi-way $\theta$-joins. The following example illustrates our setting. Assume that we wish to detect potential credit card fraud. Credit card transactions specify their timestamp ($ts$), account number ($acc$), and amount ($amnt$). A typical fraud pattern is that, in a short period of time,

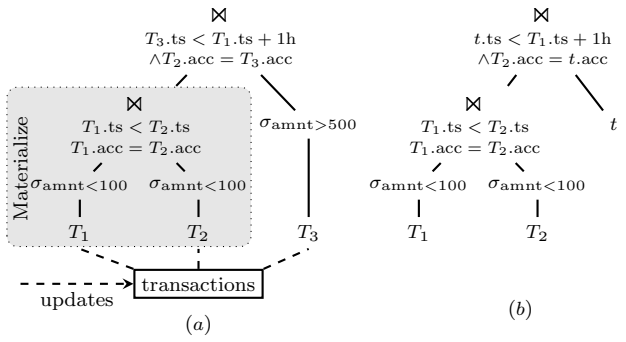


**Fig. 1** ($a$) Example query for detecting fraudulent credit card activity. ($b$) Delta query of (a) to be executed upon insertion of new high-amount transaction tuple $t$.

a criminal tests a stolen credit card with a few small purchases to then make larger purchases (cf. [37]). Assuming that the short period of time is 1 hour, this pattern could be detected by dynamically evaluating the query in Fig. 1($a$). Queries like this may exhibit arbitrary local predicates and multi-way joins with equality as well as inequality predicates.

Dynamic query evaluation has a rich history in data management, and has been researched in the context of Incremental View Maintenance (IVM) [22, 29, 30], Stream Join Processing [4,20,34,43], and Complex Event Recognition (CER, also known as Complex Event Processing) [11,15,31,46,49]). All of the existing techniques are based on recomputation of query (sub)results and/or on their materialization. We next illustrate the issues with recomputation and result materialization; a detailed literature review is given in Section 2.

The most extreme form of recomputation is of course full recomputation: simply re-evaluate $Q$ on $db + u$ whenever an update $u$ arrives on $db$. Clearly, this incurs the highest possible update processing cost, since it completely ignores the fact that $Q(db)$ was already evaluated and certain shared computation could be avoided. This can be solved by introducing the simplest form of materialization: simply store the current result $Q(db)$ and, whenever an update $u$ arrives, evaluate the *delta query* $\Delta Q$ associated to $Q$ [22]. $\Delta Q$ takes as input $db$ and $u$, and computes the update that needs to be applied to the materialized $Q(db)$ to obtain $Q(db + u)$. While this exploits certain shared computation, it unfortunately does so only to some extent.

Let us illustrate this by means of our example fraud query in Fig. 1($a$). Assume that $u$ inserts a single new high-amount transaction $t$. Then $\Delta Q(db, u)$ amounts to computing the join shown in Fig. 1($b$). While we can expect that this is more efficient that full recomputation of $Q$ (since the join with $T_3$ is replaced by the join with a single tuple), observe that if we now get another update $u'$ that inserts another high-amount transaction $t'$ then we are essentially *fully* recomputing the join of Fig. 1($b$), but now with $t'$ instead of $t$. Hence, by fully recomputing $\Delta Q$ whenever a new update $u'$ arrives, we are ignoring the fact that certain computation—most notably, the join between $T_1$ and $T_2$ common to $t$ and $t'$—is shared and in principle need not be redone.

This can be solved by introducing more materialization: in addition to storing $Q(db)$ also materialize the result of the join between $T_1$ and $T_2$ indicated in the shaded area in Fig. 1($a$), and use additional delta queries to maintain this result as well as $Q(db)$. In the IVM literature this approach is known as Higher-Order IVM (HIVM). This method is highly effective in practice, and formally lowers the update processing complexity [29].

While more materialization hence means less recomputation, it has a serious drawback in terms of additional memory overhead: materialization of $Q(db)$ requires $\Omega(|Q(db)|)$ space, where $|db|$ denotes the size of $db$. Therefore, when $Q(db)$ is large, which is often the case in CER as well as in data preparation scenarios for training statistical models, materializing $Q(db)$ quickly becomes impractical, especially for main-memory based systems. Note that $|Q(db)|$ can be polynomial in $|db|$. HIVM is even more affected by this problem since it not only materializes the result of $Q$ but also the results of partial joins, which can be larger than both $db$ and $Q(db)$. For example, the shaded area of Fig. 1(a) builds the the table of all pairs of *small* transactions that could be part of a credit card fraud. If we assume that there are $N$ small transactions, all of the same account, this materialization will take $\Theta(N^2)$ space. This naturally becomes impractical when $N$ grows.

In summary, in traditional techniques for dynamic query evaluation there is a trade-off between recomputation and materialization: more materialization means less recomputation and hence faster update processing, but more memory consumption. In previous work [25,26], we have shown that this trade-off can be avoided by taking a different approach to dynamic query evaluation: instead of materializing $Q(db)$ we can build a succinct data structure that (1) supports updates efficiently and (2) *represents* $Q(db)$ in the sense that from it we can *generate* $Q(db)$ as efficiently as if it were materialized. In particular, the representation is equipped with index structures so that we can enumerate $Q(db)$ *with bounded delay* [38]: one tuple at a time, while spending only a small amount of work to produce each new tuple. This makes the enumeration competitive with enumeration from materialized query results.

In essence, we hence separate dynamic query processing into two stages: (1) an update stage where we only maintain under updates the (small) information that is

necessary for result enumeration and (2) an enumeration stage where the query result is efficiently enumerated.

The main insight of [25,26] is that a practical family of algorithms for dynamic query evaluation based on this idea can naturally be obtained by modifying Yannakakis' seminal algorithm for processing acyclic joins in the static setting [47]. In particular, instead of materializing $Q(db)$ and its sub-joins (which can both be large) it suffices to materialize *semijoins* (which are of size linear in $db$), and build indexes on these semijoins as well as the base relations. The most general form of this modification, called *General Dynamic Yannakakis* (GDyn), supports the class of acyclic Generalized Conjunctive Queries (GCQs), which are acyclic conjunctive queries with $\theta$-joins, that are evaluated under multiset semantics and support certain forms of aggregation. The representation of query results that underlies GDyn has several desirable properties:

- It allows to enumerate $Q(db)$ with bounded delay.
- It requires only $\mathcal{O}(|db|)$ space and is hence independent of the size of $Q(db)$.
- It features efficient maintenance under updates. When $Q$ is a conjunctive query (with equijoins only), then we can update the representation of $Q(db)$ to a representation of $Q(db+u)$ in time $\mathcal{O}(|db| + |u|)$. In contrast, existing techniques may require $\Omega(|u| + |Q(db+u)|)$ time in the worst case. For the subclass of $q$-hierarchical queries [8], our update time is $\mathcal{O}(|u|)$. When $Q$ consists of both equality and inequality joins $(<, \leq)$ the update time increases. If $Q$ has at most one inequality per pair of relations, the update time is $\mathcal{O}(M \log M)$, where $M = |db| + |u|$; otherwise it is $\mathcal{O}(M^2)$. In contrast, existing techniques may require $\Omega(|db|^{k-1})$ time in the worst case, where $k$ is the number of relations to be joined.

In this paper, we consolidate and expand our development of GDyn. Our contributions are as follows.

(1) We give an intuitive, concise, and stand-alone description of GDyn, and explain the main components behind its efficiency. In addition, we provide, for the first time, full formal proof of GDyn's correctness and complexity (Section 4).

(2) Like most query evaluation algorithms, GDyn's operation is driven by the availability of a query plan. In our previous work, we have always assumed query plans to be explicitly given. In this paper, in contrast, we present, for the first time, an algorithm for computing GDyn query plans (Section 5).

(3) Finally, we our implementation of GDyn (Section 6) and present extended experimental comparison of GDyn to state of the art IVM and CER engines. We explore the full design space of queries with up to three joins. GDyn performs consistently better than the competitor systems with multiple orders of magnitude improvements in both time and memory consumption (Section 7).

We introduce the required background in Section 3, discuss related work in Section 2 and conclude in Section 8. Because of space restrictions, certain supporting material and proofs of auxiliary statements are given in the Appendix, which is available in the online supplementary material of this paper.

## 2 Related Work

**IVM.** The trade-off between materialization and recomputation explained in the Introduction is at the core of IVM [12,21,22,29,30]. IVM hence differs from GDyn as already explained in the Introduction.

**CER.** There are two approaches to CER: relational and automaton-based. Relational approaches (e.g., [31]) are similar to IVM. In contrast to the relational approaches, automaton approaches assume that event tuples are endowed with a timestamp and that the arrival order of event tuples corresponds to the timestamp order (i.e., there are no out-of-order events). They build an automaton to recognize the desired temporal patterns in the input stream. Broadly speaking, there are two automata-based recognition approaches. In the first approach, followed by [3,46], events are cached per state and once a final state is reached a search through the cached events is done to recognize the complex events. While it is no longer necessary to check the temporal constraints (e.g., $T_1.\text{ts} < T_2.\text{ts}$) during the search, the additional constraints (in our fraud query example $T_1.\text{acc} = T_2.\text{acc} = T_3.\text{acc}$) must still be verified. At essence, this corresponds to fully re-computing a delta query since each event triggering a transition to a final state may cause re-evaluation of a sub-join on the cached data. In the second approach, followed by [11,14,15,49], partial runs are materialized according to the automaton's topology. For our example query, this means that, just like HIVM, the join in the shaded area of Fig. 1($a$) is materialized and maintained so it is available when a large amount transaction arrives. This approach hence shares with HIVM its high memory overhead.

**Stream Joins.** The goal of stream join processing is to produce join results incrementally as new tuples are added to the input or old tuples retracted. To see how GDyn differs from stream joins, we discern two classes of stream join algorithms.

Algorithms in the first class are designed to produce output tuples as soon as possible, without blocking for more input to become available. They are based on

the symmetric hash-join [45] and its variants [41, 43]. Algorithms in this class favor full recomputation of delta queries, and hence ignore the opportunity to reduce redundant recomputation by additional materialization. Moreover, these algorithms are limited to processing equijoins only.

Algorithms in the second class focus on window-based stream joins [20, 27, 34, 40]. Like the automata-based approaches in CER, they assume that each tuple is endowed with a timestamp attribute, and are restricted to the setting where tuples follow a FIFO paradigm: new tuples arrive in increasing timestamp order and tuples with the oldest timestamp are deleted first. This property is crucial for the algorithm's proposed optimizations to work. GDyn, in contrast makes no FIFO assumption and can deal with arbitrary updates, including out-of-order updates. Furthermore, we note that window-based joins are a strict subclass of the class of all inequality-joins, since in a window-based join only a single temporal attribute will be compared across all relations. As such, queries like $R_1 \bowtie_{R_1.amnt<R_2.amnt} R_2 \bowtie_{R_2.ts<R_3.ts} R_3$ that inequality-join across multiple unrelated attributes are not considered by [20, 27, 34, 40]. GDyn, in contrast, processes such queries intelligently. Finally, we note that, in contrast to GDyn, support for multi-way stream joins is limited: [27, 34, 40] consider only binary joins, while [20] does treat multi-way joins, but makes the simplifying assumption that all comparisons are on the same, single attribute.

Because of the increasing wide-spread use of distributed compute engines such as Flink, Spark, and Storm in contemporary data analysis scenarios, there has also been much research on how to support stream joins in such engines (e.g., [44]). To the best of our knowledge, this work builds upon the above-mentioned centralized stream join algorithms (while tackling additional challenges such as distribution and fault-tolerance), and hence similarly differ from GDyn as described above. We leave the extension of GDyn to the parallel en distributed setting as an interesting avenue for future work.

**Query evaluation with constant delay enumeration** has gained increasing attention in the last decade [6–8, 10, 25, 32, 33, 33, 36, 38]. This setting, however, deals with equijoins only.

**Inequality joins.** Also related, although restricted to the static setting, is the practical evaluation of binary [17, 18, 23] and multi-way [9, 48] inequality joins. Our work, in contrast, considers dynamic processing of multi-way $\theta$-joins, with a specialization to inequality joins. Khayyat et al. [28] proposed fast multi-way inequality join algorithms based on sorted arrays and space efficient bit-arrays. They focus on the case where there are exactly two inequality conditions per pairwise join. While they also present an incremental algorithm for pairwise joins, their algorithm makes no effort to minimize the update cost of multi-way joins. As a result, they either materialize subresults (implying a space overhead that can be more than linear), or recompute subresults.

## 3 Preliminaries

**Query Language.** Throughout the paper, let $x, y, z, \ldots$ denote *variables* (also commonly called *column names* or *attributes*). A *hyperedge* is a finite set of variables. We use $\overline{x}, \overline{y}, \ldots$ to denote hyperedges. A *Generalized Conjunctive Query* (GCQ) is an expression of the form

$$Q = \pi_{\overline{y}} \left( r_1(\overline{x_1}) \bowtie \cdots \bowtie r_n(\overline{x_n}) \mid \bigwedge_{i=1}^{m} \theta_i(\overline{z_i}) \right). \tag{1}$$

Here $r_1, \ldots, r_n$ are *relation symbols*; $\overline{x_1}, \ldots, \overline{x_n}$ are hyperedges (of the same arity as $r_1, \ldots, r_n$); $\theta_1, \ldots, \theta_m$ are predicates over $\overline{z_1}, \ldots, \overline{z_m}$, respectively; and both $\overline{y}$ and $\bigcup_{i=1}^{m} \overline{z_i}$ are subsets of $\bigcup_{i=1}^{n} \overline{x_i}$. We treat predicates abstractly: for our purpose, a predicate over $\overline{x}$ is a (not necessarily finite) decidable set $\theta$ of tuples over $\overline{x}$. For example, $\theta(x, y) = x < y$ is the set of all tuples $(a, b)$ satisfying $a < b$. We indicate that $\theta$ is a predicate over $\overline{x}$ by writing $\theta(\overline{x})$. Throughout the paper, we consider only non-nullary predicates, i.e., predicates with $\overline{x} \neq \emptyset$.

*Example 1* The following query is hence a GCQ.

$$\pi_{x,y,z} \left( r(x, y) \bowtie s(z, u) \bowtie t(v, w) \mid x < u, z < w \right).$$

Since, as usual, the natural join between relations that have a disjoint schema is simply their cartesian product, this query asks to take the cartesian product of $r(x, y), s(z, u)$ and $t(v, w)$, subsequently select those tuples that satisfy $x < u$ and $z < w$, and finally project on $x, y, z$. Likewise, the GCQ

$$\pi_{y,z} \left( r(x, y) \bowtie s(y, z) \bowtie t(z, v) \mid y < v \right)$$

asks to take the natural join of $r(x, y), s(y, z)$ and $t(z, v)$ (where $r(x, y)$ and $s(y, z)$ equijoin on $y$, and $s(y, z)$ and $t(z, v)$ equijoin on $z$), subsequently select those tuples that satisfy $y < v$, and project on $y, z$.

We call $\overline{y}$ the *output variables* of $Q$ and denote it by *out*$(Q)$. If $\overline{y} = \overline{x_1} \cup \cdots \cup \overline{x_n}$ then $Q$ is called a *full query* and we may omit the symbol $\pi_{\overline{y}}$ altogether for brevity. We denote by *full*$(Q)$ the full *GCQ* obtained from $Q$ by setting *out*$(Q)$ to $\overline{x_1} \cup \cdots \cup \overline{x_n}$. The elements $r_i(\overline{x_i})$ are called *atoms*. *at*$(Q)$ denotes the set of all atoms in $Q$, and *pred*$(Q)$ the set of all predicates in $Q$. A *conjunctive query* (or CQ) is a GCQ where *pred*$(Q) = \emptyset$.

**Semantics.** We evaluate GCQs over Generalized Multiset Relations (GMRs for short) [25,29,30]. Let $\mathrm{dom}(x)$ denote the domain of variable $x$. As usual, a tuple over $\overline{x}$ is a function $\mathbf{t}$ that assigns a value from $\mathrm{dom}(x)$ to every $x \in \overline{x}$. $\mathbb{T}[\overline{x}]$ denotes the set of all tuples over $\overline{x}$. A GMR over $\overline{x}$ is a function $R\colon\ \mathbb{T}[\overline{x}] \to \mathbb{Z}$ mapping tuples over $\overline{x}$ to integers such that $R(\mathbf{t}) \neq 0$ for finitely many tuples $\mathbf{t}$. In contrast to classical multisets, the multiplicity of a tuple in a GMR can hence be negative, allowing to treat insertions and deletions uniformly. We write $\mathit{var}(R)$ for $\overline{x}$; $\mathrm{supp}(R)$ for the finite set of all tuples with non-zero multiplicity in $R$; $\mathbf{t} \in R$ to indicate $\mathbf{t} \in \mathrm{supp}(R)$; and $|R|$ for $|\mathrm{supp}(R)|$.

Fig. 2 illustrates the operations of GMR union ($R + S$), minus ($R - S$), projection ($\pi_{\overline{z}} R$), natural join ($R \bowtie T$) and selection ($\sigma_P(R)$), which are defined similarly as in relational algebra with multiset semantics. See [25,30] for formal semantics. We stress that, as usual, if $R$ and $T$ have disjoint schema then $R \bowtie T$ is simply their cartesian product.

A GMR $R$ is *positive* if $R(\mathbf{t}) > 0$ for all $\mathbf{t} \in \mathrm{supp}(R)$. A *database* over a set $\mathcal{A}$ of atoms is a function $db$ that maps every atom $r(\overline{x}) \in \mathcal{A}$ to a positive GMR $db_{r(\overline{x})}$ over $\overline{x}$. Given a database $db$ over the atoms occurring in query $Q$, the evaluation of $Q$ over $db$, denoted $Q(db)$, is the GMR over $\overline{y}$ constructed in the expected way: take the natural join of all GMRs in the database, do a selection over the result w.r.t. each predicate, and finally project on $\overline{y}$. It is instructive to note that after evaluation, each result tuple has an associated multiplicity that counts the number of derivations for the tuple. In other words, the query language has built-in support for `COUNT` aggregations. We note that, in their full generality, GMRs can carry multiplicities that are taken from an arbitrary algebraic semiring structure (cf., [29]), which can be useful to describe the computation of more advanced aggregations over the result of a GCQ [2]. To keep the notation and discussion simple we fix the ring $\mathbb{Z}$ of integers throughout the paper, but our results generalize to arbitrary semirings and their associated aggregations.

**Semijoins.** If $\Theta$ is a set of predicates then we write $\sigma_\Theta R$ for $\sigma_{\bigwedge_{\theta\in\Theta}} R$ and $R \ltimes_\Theta S$ for $\sigma_\Theta(R \bowtie S)$. If $\overline{z} \subseteq \mathit{var}(R)$ or $\overline{z} \subseteq \mathit{var}(S)$ then $\pi_{\overline{z}}(R \ltimes_\Theta S)$ is called a *semijoin*. We write $R \ltimes_\Theta S$ for the subGMR of $R$ consisting of all tuples that have a joining tuple in $S$:

$$R \ltimes_\Theta S \in \mathbb{T}[\mathit{var}(R)] \to \mathbb{Z}\colon \quad \mathbf{t} \mapsto \begin{cases} R(\mathbf{t}) & \text{if } \mathbf{t} \in \pi_{\mathit{var}(R)}(R \ltimes_\Theta S) \\ 0 & \text{otherwise} \end{cases}$$

Often, $S$ will contain only a single tuple $\mathbf{t}$ with multiplicity 1. In that case we simply write $R \ltimes_\Theta \mathbf{t}$.

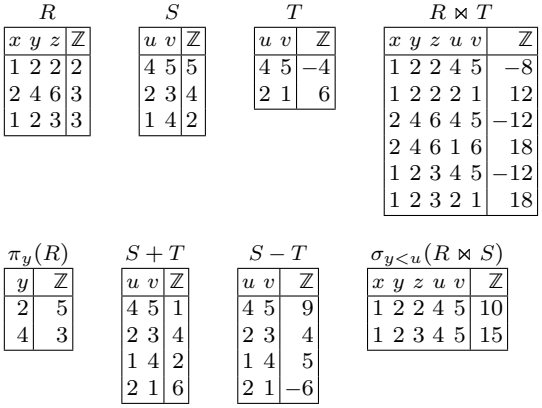


**Fig. 2** Operations on GMRs

**Updates.** An *update* to a GMR $R$ is simply a GMR $\Delta R$ over the same variables as $R$. Applying update $\Delta R$ to $R$ yields the GMR $R + \Delta R$. An *update to a database db* is a collection $u$ of (not necessarily positive) GMRs, one GMR $u_{r(\overline{x})}$ for every atom $r(\overline{x})$ of $db$, such that $db_{r(\overline{x})} + u_{r(\overline{x})}$ is positive.[1] We write $db + u$ for the database obtained by applying $u$ atom-wise to $db$.

**Computational Model.** We focus on dynamic query evaluation in main-memory. We assume a model of computation where the space used by tuple values and integers, the time of arithmetic operations on integers, the time of operations on tuples (such as projecting a tuple on a subset of its variables, or taking the union of two tuples) and the time of memory lookups are all $\mathcal{O}(1)$. We further assume that hash tables have $\mathcal{O}(1)$ access and update times while requiring linear space. While it is well-known that real hash table access is $\mathcal{O}(1)$ *expected* time and updates are $\mathcal{O}(1)$ *amortized*, complexity results that we establish for this simpler model can be expected to translate to average (amortized) complexity in real-life implementations [13].

A direct consequence of these assumptions is that, using standard database implementation techniques, every GMR $R$ can be represented in our model by a data structure that allows (1) enumeration of $R$ with delay $\mathcal{O}(1)$ (as defined in Section 4.1); (2) multiplicity lookups $R(\mathbf{t})$ in $\mathcal{O}(1)$ time given $\mathbf{t}$; (3) single-tuple insertions and deletions in $\mathcal{O}(1)$ time; while (4) having size that is proportional to $|R|$. In addition we assume it possible to sort GMRs by a given order on its tuples in $\mathcal{O}(|R| \log |R|)$ time, after which it allows enumeration in the given order with $\mathcal{O}(1)$. Single-tuple insertions that keep the GMR sorted become $\mathcal{O}(\log |R|)$ in this case.

[1] Note that, in this framework, value modifications inside a tuple are modeled by deleting the tuple with the old value, and then re-inserting the tuple, but now with the new value.

## 4 General Dynamic Yannakakis

In this section we formulate GDyn, a dynamic version of the Yannakakis algorithm [47], that focuses on the evaluation of *GCQs*. GDyn takes a non-standard approach to dynamic query evaluation: instead of materializing $Q(db)$ and sub-joins of $Q$, GDyn builds a succinct, efficiently updatable data structure that represents $Q(db)$ in the sense that from it we can *enumerate* $Q(db)$. Formally, a data structure $D$ supports *enumeration* of a set $E$ if there is a routine ENUM such that ENUM($D$) outputs each element of $E$ exactly once. Such enumeration occurs with delay $d$ if the time until the first element is output; the time between any two consecutive elements; and the time between the last element and the termination of ENUM($D$), are all bounded by $d$. $D$ supports enumeration of a GMR $R$ if it supports enumeration of the set $E_R = \{(\mathbf{t}, R(\mathbf{t})) \mid \mathbf{t} \in \text{supp}(R)\}$.

When evaluating a GCQ $Q$ over a database $db$, we will be interested in representing the elements of $Q(db)$ by means of a data structure $D_{db}$, such that we can enumerate $Q(db)$ from $D_{db}$. If, for every $db$, the delay to enumerate $Q(db)$ from $D_{db}$ is sublinear in $|db|$ then we say that the enumeration occurs with *sublinear delay*. Similarly, if the delay is independent of $|db|$ then we say that the enumeration occurs *with constant delay* [38]. Note that this is sublinear/constant in data complexity [42]: the delay may still depend on the size of the query $Q$. This is reasonable since $Q$ specifies the arity of the query result, and a larger arity inherently implies a longer delay between elements. Note that enumeration with constant delay is what we typically obtain by materializing $Q(db)$. For example, if we store the elements of $Q(db)$ in an array then enumerating $Q(db)$ amounts to scanning the array where each element access is $\mathcal{O}(1)$.

We start the development of GDyn by first giving some intuition into Constant Delay Enumeration (henceforth: CDE) in Section 4.1. The algorithm itself is stated in Section 4.2, while proofs of its correctness and analysis of its complexity is given in Section 4.3. Finally, we specialize GDyn to the case where all $\theta$-joins are inequality joins in Section 4.4.

### 4.1 Intuition

In this section, we discuss how we can obtain CDE of the result $Q(db)$ of a GCQ $Q$. Of course, the simplest way to obtain this, is simply to materialize $Q(db)$. Unfortunately, this requires memory proportional to $|Q(db)|$ which, depending on $Q$, can be of size polynomial in $|db|$. We hence desire other data structures to represent $Q(db)$ using less space, while still allowing CDE. Let us build some intuition on how this can be done by subsequently considering three queries of increasing complexity:

$$Q_1 = r(x,y) \bowtie s(y,z),$$
$$Q_2 = s(y,z) \bowtie t(z,v) \mid v < y, \quad \text{and}$$
$$Q_3 = r(x,y) \bowtie s(y,z) \bowtie t(z,v) \mid v < y.$$

Throughout our discussion assume that the GMRs assigned to $r(x,y), s(y,z)$, and $t(z,v)$ by input database $db$ are $R, S$, and $T$, respectively.

It is instructive to start with the simple binary equijoin query $Q_1 = r(x,y) \bowtie s(y,z)$ and analyze why traditional join processing algorithms do not yield CDE. Suppose that we evaluate $Q_1$ using a simple in-memory hash join with $R$ as probe relation and $S$ as build relation. Assume that the corresponding hash index of $S$ on $y$ has already been computed. Concretely, this hash index allows us to retrieve, for every $y$-tuple $\mathbf{t}$, in $\mathcal{O}(1)$ time a pointer to the GMR $S \ltimes \mathbf{t}$ of all $S$-tuples that join with $\mathbf{t}$. Now observe that, when iterating over $R$ to probe the index, we may have to visit an unbounded number of $R$-tuples that do not join with any $S$-tuple. Consequently, the delay between consecutive outputs may be as large as $|R|$. A similar analysis shows that other join algorithms, such as the sort-merge join, do not yield CDE.

How then can we obtain CDE for $r(x,y) \bowtie s(y,z)$? Intuitively, if we can ensure to only iterate over those $R$-tuples that have matching $S$-tuples, we trivially obtain constant delay since then every probe will yield a new output tuple. As such, the key is to first compute the semijoin $\rho_{x,y} = \pi_{x,y}(R \bowtie S)$. We can then iterate over the elements of $\rho_{x,y}$, probing $S$ in each iteration to generate the output with constant delay. Note that, because $\rho_{x,y}$ is a semijoin, the space needed to store $\rho_{x,y}$ is linear in $|db|$.

CDE for queries that feature $\theta$-joins can be obtained similarly. Consider $Q_2 = (s(y,z) \bowtie t(z,v) \mid y < v)$ which is a combination of an equijoin on $z$ and inequality join on $y < v$. To obtain CDE for $Q_2$, first compute the semijoin $\rho_{y,z} = \pi_{y,z}(S(y,z) \bowtie_{y<v} T(y,v))$ which consists of all tuples in $S$ that have a matching tuple in $T$. Assume for a moment that we have a more powerful index structure $I$ that allows, for any $\{y,z\}$-tuple $\mathbf{s}$, to enumerate $T \ltimes_{y<v} \mathbf{s}$ with constant delay. We can then obviously enumerate $Q_2(db)$ with constant delay by iterating over $\mathbf{s} \in \rho_{y,z}$, and for each such $\mathbf{s}$, probe $I$ to produce the tuples $\mathbf{t} \in T \ltimes_{y<v} \mathbf{s}$, outputting $(\mathbf{s} \cup \mathbf{t}, S(\mathbf{s}) \times T(\mathbf{t}))$ for each such $\mathbf{s}$ and $\mathbf{t}$. Since $T \ltimes_{y<v} \mathbf{s}$ allows CDE and multiplicity lookups are $\mathcal{O}(1)$, the entire procedure is CDE. The key question then, is how we can build this more powerful index structure $I$. The solution is to group $T$ on $z$; subsequently sort each group in descending order on $v$; and create a normal (hash-based)

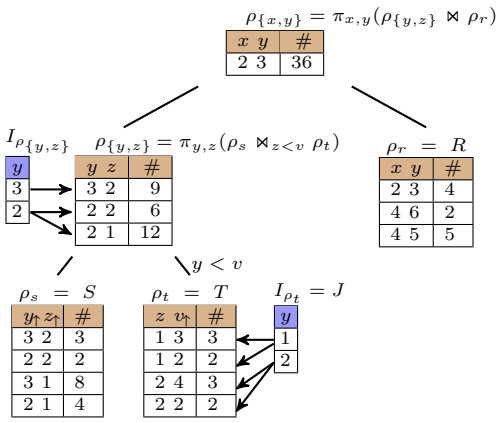


**Fig. 3** Illustration of query $Q_3$.

index $J$ then allows to find the group for each $z$ value. This now supports CDE of $T \ltimes_{y<v} \mathbf{s}$: first, use $J$ to get a pointer to the group with $z$-value $\mathbf{s}(z)$ in $\mathcal{O}(1)$ time, and then enumerate this group with constant delay and in decreasing order on $v$. Yield the current tuple $\mathbf{t}$ that is being enumerated in this fashion, provided that $\mathbf{s}(y) < \mathbf{t}(v)$. As soon as $\mathbf{s}(y) \geq \mathbf{t}(v)$ we know that all subsequent $\mathbf{t}$ will fail the inequality, and we can hence terminate. The lower left of Fig. 3 illustrates $S$, $T$, $\rho_{y,z}$, and $J$.

CDE for queries that join more than two relations can be obtained similarly, but now by computing nested semijoins. Fig. 3 illustrate how to obtain CDE for $Q_3$. Concretely, we first ensure CDE of the subquery $Q_2$ of $Q_3$ as already explained above: by computing the semijoin $\rho_{y,z} = \pi_{y,z}(S(y,z) \ltimes_{y<v} T(y,v))$, and suitably indexing $T$. Then CDE of $Q_3(db)$ is obtained by observing that $Q_3(db) \equiv R \bowtie Q_2(db)$, where $R \bowtie Q_2(db)$ is a binary equi-join, which can hence treated completely analogous as $Q_1$. Concretely, compute the nested semijoin $\rho_{x,y} = \pi_{x,y}(R \ltimes \rho_{y,z})$, and build a hash index of $\rho_{y,z}$ on $y$. (This index is depicted as $I_{\rho_{y,z}}$ in Fig. 3). Enumeration of $Q_3(db)$ is done by iterating over the tuples $\mathbf{r} \in \rho_{x,y}$, and for each such tuple $\mathbf{r}$, use $I_{\rho_{y,z}}$ to get a pointer to $\rho_{y,z} \ltimes \mathbf{r}$, which consists of all $\mathbf{s} \in \rho_{y,z}$ that equi-join with $\mathbf{r}$. Iterate over these $\mathbf{s}$ with constant delay, and finally use the more advanced index on $T$ to enumerate all tuples $\mathbf{t} \in T \ltimes_{y<v} \mathbf{s}$. For each such $\mathbf{r}, \mathbf{s}$, and $\mathbf{t}$, we output $(\mathbf{r} \cup \mathbf{s} \cup \mathbf{t}, R(\mathbf{r}) \times S(\mathbf{s}) \times T(\mathbf{t}))$. By construction of $\rho_{x,y}$ we are ensured that matching $\mathbf{s}$ will exists for every $\mathbf{r}$. Similarly, matching $\mathbf{t}$ exist for every $\mathbf{s}$ by construction of $\rho_{y,z}$. Therefore, each tuple that we iterate over will produce a new output, and the entire enumeration of $Q(db)$ is CDE.

**In conclusion.** As the examples above illustrate, we can obtain CDE for GCQs by computing (nested) semijoins, and suitably indexing both base relations and semijoin results for enumeration. Because the only additional relations that we compute are obtained by semijoining existing relations, the size of all additional GMRs that are stored is *linear* in the input *db*. Contrast this to techniques that materialize subjoin results, whose size may become polynomial in the database.

**Updates.** We finish this section by remarking that, in the presence of updates, this approach is only valid if we materialize and maintain all required semijoin results. To speed up the maintenance of semijoin results under updates, it is sometimes beneficial to create additional indexes that help in incremental computation of the semijoins, as we illustrate next. Reconsider, $Q_3$ as illustrated in Fig. 3. If we receive an update $\Delta T$ to $T$ then we need to correspondingly update $\rho_{y,z}$ from $\pi_{y,z}(S \ltimes_{y<v} T)$ to $\pi_{y,z}(S \ltimes_{y<v} (T + \Delta T))$. To that end, it suffices to compute $\Delta\rho_{y,z} = \pi_{y,z}(S \ltimes_{y<v} \Delta T)$, and add this to $\rho_{y,z}$. Computing $\Delta\rho_{y,z}$ by means of a nested loop join has $\Omega(|S| \times |\Delta T|)$ complexity. We can do better if we index $S$ by sorting $S$ lexicographically on $(z,y)$, in decreasing order. (This is actually how $S$ is depicted in Fig. 3.) $\Delta\rho'_{y,z}$ can then be computed by means of a hybrid form of sort-merge and index nested loop join. First, group $\Delta T$ on $z$ and, per group, sort tuples in decreasing order on variable $v$. Create a hash index on $\Delta T$ to be able to quickly find each group by a given $z$ value. Second, iterate over the tuples in $S$ in the given lexicographic order. For each $z$-group in $S$, find the corresponding group in $\Delta T$ by passing the $z$-value to the hash table. Let $\mathbf{s}$ be the first tuple in the $S$ group. Then iterate over the tuples of the $\Delta T$ group in decreasing order on $v$, and sum up their multiplicities until $\mathbf{s}(y)$ becomes larger than $v$. Add $\mathbf{s}$ to $\Delta\rho_{y,z}$, with its original multiplicity in $S$ multiplied by the found sum (provided that it is non-zero). Then consider the next tuple in the $S$-group, and continue summing from the current tuple in the $\Delta T$ group until $\mathbf{s}(y)$ becomes again larger than $v$, and add the result tuple with the correct multiplicity. Continue repeating this process for each tuple in the $S$ group, and for each group in $S$. Assuming that the index on $S$ already existed, then the total cost of computing $\rho_{y,z}$ in this way is $\mathcal{O}(|S| + |\Delta T| + |\Delta T| \log |\Delta T|)$ since we scan $S$ and $\Delta T$ only once, need to sort $\Delta T$, and create a hash table for each group. This is much better than the $\mathcal{O}(|S| \times |\Delta T|)$ complexity of a nested loop.

## 4.2 The Algorithm

We now turn to the general formulation of the Dynamic Yannakakis algorithm. As exemplified in Section 4.1, GDyn obtains CDE by computing (nested) semijoins, and indexing both these semijoins and base relations.

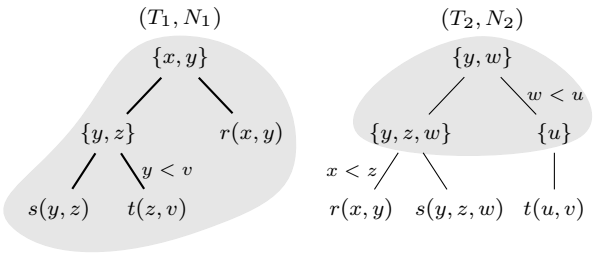


**Fig. 4** Two example plans. The connex sets are indicated by the shaded areas.

The order in which semijoins are computed, and how they are indexed is recorded in a *dynamic query plan*, which is introduced next.

**Dynamic Query Plans.** To simplify notation, we denote the set of all variables (resp. atoms, resp. predicates) that occur in an object $X$ (such as a query) by $var(X)$ (resp. $at(X)$, resp. $pred(X)$). In particular, if $X$ is itself a set of variables, then $var(X) = X$. We extend this notion uniformly to labeled trees. E.g., if $n$ is a node in tree $T$, then $var_T(n)$ denotes the set of variables occurring in the label of $n$, and similarly for edges and trees themselves. If $T$ is clear from the context, we omit subscripts from our notation.

**Definition 1** A *Dynamic Query Plan* (or simply: *plan*) is a tuple $(T, N)$ where $T$ is a binary generalized join tree, and $N$ is a sibling-closed connex subset of $T$. A *Generalized Join Tree* (GJT) is a node-labeled and edge-labeled directed tree $T = (V, E)$ such that:

- Every leaf is labeled by an atom.
- Every interior node $n$ is labeled by a hyperedge and has at least one child $c$ such that $var(n) \subseteq var(c)$. Such a child is called a *guard* of $n$.
- Whenever the same variable $x$ occurs in the label of two nodes $m$ and $n$ of $T$, then $x$ occurs in the label of each node on the unique path linking $m$ and $n$. This condition is called the *connectedness condition*.
- Every edge $p \to c$ from parent $p$ to child $c$ in $T$ is labeled by a set $pred(p \to c)$ of predicates. It is required that for every predicate $\theta(\overline{z}) \in pred(p \to c)$ we have $var(\theta) = \overline{z} \subseteq var(p) \cup var(c)$.

$T$ is *binary* if every node in $T$ has at most two children. A *connex subset* of $T$ is a set $N \subseteq V$ that includes the root of $T$ such that the subgraph of $T$ induced by $N$ is a tree. $N$ is *sibling-closed* if for every node $n \in N$ with a sibling $m$ in $T$, $m$ is also in $N$. The *frontier* of a connex set $N$ is the subset $F \subseteq N$ consisting of those nodes in $N$ that are leaves in the subtree of $T$ induced by $N$.

Fig. 4 shows two plans $(T_1, N_1)$ and $(T_2, N_2)$. The set $N_1$ contains all nodes of $T_1$, and is therefore a sibling-closed connex subset of $T_1$. Its frontier is $\{r(x, y), s(y, z), t(z, v)\}$. If we remove $\{y, z\}$ from $N_1$ the set is no longer connex. If instead we remove $r(x, y)$ the set is still connex, but no longer sibling-closed. The set $N_2$ is a sibling-closed connex subset of $T_2$, and its frontier is $\{\{y, z, w\}, \{u\}\}$. Removing either $\{y, z, w\}$ or $\{u\}$ makes $N_2$ no longer sibling-closed.

**Definition 2** Let $T$ be a GJT and let $N$ be a connex subset of $T$. Assume that $\{\!|r_1(\overline{x_1}), \ldots, r_n(\overline{x_n})|\!\}$ is the multiset of atoms occurring as labels in the leaves of $T$. Then the query associated to $T$ is the full join

$$\mathcal{Q}[\![T]\!] = r_1(\overline{x_1}) \bowtie \cdots \bowtie r_n(\overline{x_n}) \mid \bigwedge_{\theta(\overline{z}) \in pred(T)} \theta(\overline{z}),$$

and the query associated to $(T, N)$ is the GCQ $\mathcal{Q}[\![T, N]\!] = \pi_{var(N)}(\mathcal{Q}[\![T]\!])$.

To illustrate, referring to Fig. 4, we have $\mathcal{Q}[\![T_1, N_1]\!] = \pi_{x,y,z,v}\,(r(x, y) \bowtie s(y, z) \bowtie t(z, v) \mid y < v)$, which is $Q_3$ from Section 4.1.

**The data structure.** Following the intuition of Section 4.1, the GJT $T$ of a plan $(T, N)$ specifies the semijoin results that need be maintained and indexed, while the connex set $N$ drives the enumeration of query results. We formalize this next. Because in this section we are introducing GDyn for arbitrary GCQs (with arbitrary join predicates $\theta$), we first need to introduce general notions of an index.

**Definition 3 (Enumeration Index)** Let $R$ be a GMR, $\theta$ a predicate, and $\overline{y}$ be a hyperedge. A datastructure $I$ that is of size linear in $R$ and that allows, for any given $\overline{y}$-tuple $\mathbf{t}$, enumeration of $(R \ltimes_\theta \mathbf{t})$ with delay $\mathcal{O}(f(|R|))$ is called an *enumeration index of $R$ by $(\theta, \overline{y})$ with delay* $f\colon \mathbb{N} \to \mathbb{N}$.

For example, in Section 4.1 we have discussed how, by means of grouping and sorting, we can obtain an enumeration index of $T(z, v)$ on $(y < v, \{y, z\})$ with constant delay.

**Definition 4 (Join Index)** Let $R$ be a GMR, $\theta$ be a predicate, and $\overline{y}, \overline{z}$ be hyperedges such that $\overline{z} \subseteq var(R)$ or $\overline{z} \subseteq \overline{y}$. A datastructure $I$ that is of size linear in $R$ and that allows, for any GMR $S$ over $\overline{y}$ computation of $\pi_{\overline{z}}(R \bowtie_\theta S)$ in time $\mathcal{O}(g(|R|, |S|))$ is called a *join index of $R$ by $(\theta, \overline{y}, \overline{z})$ with access time* $g\colon \mathbb{N}^2 \to \mathbb{N}$.

For example, in Section 4.1 we have discussed how, by means of grouping and sorting, we can obtain a join index of $R(y, z)$ by $(y < v, \{z, v\}, \{y, z\})$ whose access time is $\mathcal{O}(|R| + |S| \log |S|)$.

For both enumeration and join indexes, the *update time* is the time required to update the index to a corresponding (enumeration, resp. join) index on $R + \Delta R$, given update $\Delta R$ to $R$.

**Definition 5** Let $(T, N)$ be a plan and let $db$ be a database over $at(T)$. The *$T$-reduct* (or *semi-join reduction*) of $db$ is a collection $\rho$ of GMRs, one GMR $\rho_n$ for each node $n \in T$, defined inductively as follows. Let $pred(n)$ denote the set of all predicates on the edges from $n$ to its children in $T$.

- if $n = r(x)$ is an atom, then $\rho_n = db_{r(x)}$
- if $n$ has a single child $c$, then $\rho_n = \pi_{var(n)}\sigma_{pred(n)}\rho_c$
- otherwise, $n$ has two children $c_1$ and $c_2$. In this case we have $\rho_n = \pi_{var(n)}\sigma_{pred(n)}(\rho_{c_1} \bowtie \rho_{c_2})$. Note that, because $n$ has a guard child, this is actually a semijoin.

A $T$-reduct needs to be augmented by suitable index structures to be used for both enumeration and maintenance under updates. Concretely for each node $n$ with parent $p$ in $T$, the following indexes are created:

- If $n$ belongs to $N$, then we store an enumeration index $P_n$ on $\rho_n$ by $(pred(p \to n), var(p))$.
- If $n$ is a node with a sibling $m$, then we store a join index $S_n$ on $\rho_n$ by $(pred(p), var(m), var(p))$.

The $T$-reduct $\rho$ together with the collection of indexes is called a *$(T, N)$-representation* for $db$, or $(T, N)$-rep for short.

Reconsider the plan $(T_1, N_1)$ from Fig. 4. Fig. 3 depicts an example $(T_1, N_1)$-representation $\rho$ for the database $db$ composed of the GMRs shown at the leaves of the tree. $I_{\rho_{y,z}}$ and $I_{\rho_t}$ illustrate the enumeration indexes; the join indexes are not illustrated.

It is important to observe that, since a $T$-reduct constructs only semijoins of database GMRs, and projections thereof, each $|\rho_n|$ is linear in the size of $db$. Consequently, the indexes are also of size linear in $db$ and hence the entire $(T, N)$-rep is linear in $db$.

**Proposition 1** *$|\rho_n| \leq \max_{r(\overline{x}) \in at(T)} |db_{r(\overline{x})}|$, for every $n \in T$.*

Given these definitions, the enumeration and maintenance algorithms that form GDyn are shown in Algorithm 1. They operate as follows.

**Enumeration.** To enumerate from a $(T, N)$-rep we iterate over the reductions $\rho_n$ with $n \in N$ in a nested fashion, starting at the root and proceeding top-down. When $n$ is the root, we iterate over all tuples in $\rho_n$. For every such tuple $\mathbf{t}$, we iterate only over the tuples in the children $c$ of $n$ that are compatible with $\mathbf{t}$ (i.e., tuples in $\rho_c$ that join with $\mathbf{t}$ and satisfiy $pred(n \to c)$). Note that such tuples can be enumerated efficiently thanks to the enumeration index $P_c$. This procedure continues until we reach nodes in the frontier of $N$ at which time the output tuple can be constructed. The pseudocode is given by the routine ENUM in Algorithm 1.

**Update processing.** To maintain a $(T, N)$-rep under update $u$ it suffices to traverse the nodes of $T$ in a bottom-up fashion. At each node $n$ we have to compute the update $\Delta_n$ to apply to $\rho_n$ and its associated indexes. For leaf nodes, this update is given by the update $u$ itself. For interior nodes, $\Delta_n$ can be computed from the update and the original reduct of its children. UPDATE in Algorithm 1 gives the pseudocode. Here, line 21 is implemented by using the join index $S_m$ on $\rho_m$ by $(pred(p), var(n), var(p))$. Line 23 can be implemented by a straightforward hash-based aggregation. As a side effect of modifying $\rho$ the associated indexes are also updated (not shown).

**Algorithm 1** GDyn: General Dynamic Yannakakis

```
 1: function ENUM_{T,N}(ρ)
 2:   for each t ∈ ρ_{root(T)} do ENUM_{T,N}(root(T), t, ρ)
 3: function ENUM_{T,N}(n, t, ρ)
 4:   if n is in the frontier of N then yield (t, ρ_n(t))
 5:   else if n has one child c then
 6:     for each s ∈ ρ_c ⋉_{pred(n→c)} t do ENUM_{T,N}(c, s, ρ)
 7:   else n has two children c_1 and c_2
 8:     for each t_1 ∈ ρ_{c_1} ⋉_{pred(n→c_1)} t do
 9:       for each t_2 ∈ ρ_{c_2} ⋉_{pred(n→c_2)} t do
10:         for each (s_1, μ) ∈ ENUM_{T,N}(c_1, t_1, ρ) do
11:           for each (s_2, ν) ∈ ENUM_{T,N}(c_2, t_2, ρ) do
12:             yield (s_1 ∪ s_2, μ × ν)
13: procedure UPDATE_{T,N}(ρ, u)
14:   for each n ∈ leafs(T) labeled by r(x̄) do
15:     Δ_n ← u_{r(x̄)}
16:   for each n ∈ nodes(T) \ leafs(T) do
17:     Δ_n ← empty GMR over var(n)
18:   for each n ∈ nodes(T), traversed bottom-up do
19:     ρ_n += Δ_n
20:     if n has a parent p and a sibling m then
21:       Δ_p += π_{var(p)} (ρ_m ⋉_{pred(p)} Δ_n)
22:     else if n has parent p then
23:       Δ_p += π_{var(p)} σ_{pred(p)} Δ_n
```

## 4.3 Correctness and Complexity

We next prove correctness of the enumeration and update procedures, and bound their complexity. We start with enumeration. Throughout this section, let $(T, N)$ be a plan, let $db$ be a database over $at(T)$, and assume that we have $(T, N)$-rep of $db$ with $T$-reduct $\rho$. Given a node $n \in T$ we denote the subtree of $T$ rooted at $n$ by $T_n$, and the subset of all nodes of $N$ that are in $T_n$ by $N_n$. The following lemma relates the GMRs at each node $n$ of a $T$-reduct to the query induced by the subtree of $T$ at $n$. Here, we write $\mathcal{Q}[T_n, n]$ as a shorthand for $\mathcal{Q}[T_n, \{n\}]$. Recall from Definition 2 that this is the query that joins all atoms in $T_n$ (w.r.t. all predicates in $T_n$), and subsequently projects on $var(n)$.

**Lemma 1** *$\rho_n = \mathcal{Q}[T_n, n](db)$, for every node $n \in T$.*

The proof by induction is detailed in Appendix A. To show correctness of enumeration, we need the following additional lemma regarding the subroutine of Algorithm 1 (Line 3). The proof is again by induction and detailed in Appendix A.

**Lemma 2** *For every node $n \in N$ and every tuple $\mathbf{t}$ in $\rho_n$, $\text{ENUM}_{T,N}(n, \mathbf{t}, \rho)$ enumerates $\mathcal{Q}[\![T_n, N_n]\!](db) \ltimes \mathbf{t}$.*

Finally, we require the following insights, also proved in Appendix A.

**Lemma 3** *1. $Q(db)$ is a positive GMR, for any GCQ $Q$ and any database $db$.*
*2. If $R$ is a positive GMR over $\overline{x}$ and $\overline{y} \subseteq \overline{x}$, then $\mathbf{t}[\overline{y}] \in \pi_{\overline{y}} R$ for every tuple $\mathbf{t} \in R$.*

We note that item (2) is not true for when $R$ has both positive and negative multiplicities, since multiplicities of opposite sign could cancel each other out when projecting, thereby removing $\mathbf{t}[\overline{y}]$ from $\pi_y(R)$.

**Proposition 2** *If $\rho$ is a $T$-reduct of $db$ then $\text{ENUM}_{T,N}(\rho)$ enumerates $\mathcal{Q}[\![T, N]\!](db)$.*

*Proof.* Let $r$ be the root of $T$. By Lemma 1 we have $\rho_r = \mathcal{Q}[\![T_r, r]\!](db) = \mathcal{Q}[\![T, r]\!](db) = \pi_{var(r)}\mathcal{Q}[\![T]\!](db)$. Furthermore, $\pi_{var(r)}\mathcal{Q}[\![T]\!](db) = \pi_{var(r)}\pi_{var(N)}\mathcal{Q}[\![T]\!](db)$ since $var(r) \subseteq var(N)$ as $r \in N$. Therefore, $\rho_r = \pi_{var(r)}\pi_{var(N)}\mathcal{Q}[\![T]\!](db) = \pi_{var(r)}\mathcal{Q}[\![T, N]\!](db)$. We conclude that $\rho_r$ is a projection of $\mathcal{Q}[\![T, N]\!](db)$, and hence by Lemma 3 that every tuple in $\mathcal{Q}[\![T, N]\!](db)$ has a compatible tuple in $\rho_r$. As such, $\mathcal{Q}[\![T, N]\!](db)$ equals the disjoint union $\bigcup_{\mathbf{t} \in \rho_r} \mathcal{Q}[\![T, N]\!](db) \ltimes \mathbf{t}$. By Lemma 2, this is exactly what $\text{ENUM}_{T,N}(\rho)$ enumerates. □

We now analyze the complexity of $\text{ENUM}_{T,N}$. First, observe that by definition of $T$-reducts, it is the case that for every node $n$ and every $\mathbf{t} \in \rho_n$ there exists a tuple in $\rho_c \ltimes_{pred(n \to c)} \mathbf{t}$. Hence, every tuple that we iterate over will eventually produce a new output tuple. This ensures that we do not risk wasting time in iterating over tuples that in the end yield no output. As such, the time needed for $\text{ENUM}_{T,N}(\rho)$ to produce a single new output is dominated by the time taken to iterate over the tuples in $\rho_n \ltimes_{pred(p \to n)} \mathbf{t}$, where $p$ is the parent of $n$. Since we can use the enumeration index $P_n$ to do so efficiently, the efficiency of the entire enumeration will depend on the delay incurred by accessing the enumeration indexes. The following proposition formalizes this insight.

**Proposition 3** *Assume that every enumeration index has enumeration delay $f$, where $f$ is a monotone function. Then, using these indexes, $\text{ENUM}_{T,N}(\rho)$ enumerates $\mathcal{Q}[\![T, N]\!](db)$ with delay $\mathcal{O}(|N| f(M))$ where $M = \max_{r(\overline{x}) \in at(T)} |db_{r(\overline{x})}|$. Thus, the total time required to execute $\text{ENUM}_{T,N}(\rho)$ is $\mathcal{O}(|\mathcal{Q}[\![T, N]\!](db)||N| f(M))$.*

*Proof.* $\text{ENUM}_{T,N}(\rho)$ correctly enumerates $\mathcal{Q}[\![T, N]\!](db)$ by Proposition 2. As such, it suffices to show that that the delay satisfies the given bounds. To that end, we show that for every $n \in N$ and $\mathbf{t} \in \rho_n$, the call $\text{ENUM}_{T,N}(n, \mathbf{t}, \rho)$ produces outputs with delay $\mathcal{O}(|N_n| f(M))$. We proceed by induction on $|N_n|$. If $|N_n| = 1$ then $n$ is in the frontier of $N$ and the delay is clearly constant as the algorithm will only yield $(\mathbf{t}, \rho_n(\mathbf{t}))$ (line 4). Now assume that $|N_n| > 1$. Then $n$ is not in the frontier of $N$. If $n$ has a single child $c$, then line 6 is executed, and the enumeration index $P_c$ allows us to iterate over $\rho_c \ltimes_{pred(n)} \mathbf{t}$ with delay $\mathcal{O}(f(|\rho_c|))$, which is $\mathcal{O}(f(M))$ by Proposition 1. For each element $\mathbf{s}$ of this enumeration, the algorithm calls $\text{ENUM}_{T,N}(c, \mathbf{s}, \rho)$, which by induction hypothesis produces output elements with delay $\mathcal{O}(|N_c| f(M))$. Hence, the maximum delay between two outputs is $\mathcal{O}\left(f(|M|) + |N_c| f(M)\right) = \mathcal{O}\left((|N_c| + 1) f(M)\right) = \mathcal{O}\left(|N_n| f(M)\right)$. For the case in which $n$ has two children $c_1$ and $c_2$, lines 7–12 are executed. By similar reasoning it is easy to show that the maximum delay between outputs is

$$\mathcal{O}(f(|M|)) + \mathcal{O}(|N_{c_1}| f(M)) + \mathcal{O}(f(|M|)) + \mathcal{O}(|N_{c_2}| f(M))$$

which is bounded by $\mathcal{O}((|N_{c_1}| + |N_{c_2}| + 2) f(M)) = \mathcal{O}(2|N_n| f(M)) = \mathcal{O}(|N_n| f(M))$. □

In particular, if all enumeration indexes are with constant delay (i.e., $f(M) = \mathcal{O}(1)$), then GDyn enumerates $\mathcal{Q}[\![T, N]\!](db)$ with delay $\mathcal{O}(|N|)$, which is also constant in data complexity.

**Update processing.** We next turn our attention to the update procedure UPDATE of Algorithm 1. Since this is straightforward to prove correct, we focus on its complexity. Since UPDATE uses the join indexes available in the $(T, N)$-rep during its execution we will hence bound the running time of UPDATE in terms of the join index access and update times. We first require the following insight, which bounds the running time under the condition that the GMRs computed by UPDATE have a certain bounded size. The proof is in Appendix A.

**Proposition 4** *Assume that all join indexes in the $(T, N)$-rep have access time $g$, and that all indexes (join and enumeration) have update time $h$, where $g$ and $h$ are monotone functions. Further assume that, during the entire execution of* UPDATE*, $K$ and $U$ bound the size of $\rho_n$, resp. $\Delta_n$, for all $n$. Then, $\text{UPDATE}_{T,N}(\rho, u)$ runs in time $\mathcal{O}\left(|T| \cdot (U + h(K, U) + g(K, U))\right)$.*

We next bound the size of $\rho_n$ and $\Delta_n$ throughout the execution of UPDATE.

**Proposition 5** *During the entire execution of* UPDATE *we have* $|\rho_n| \leq M$ *and* $|\Delta_n| \leq 4M$ *for every* $n \in T$*, where* $M = \max_{r(\overline{x}) \in at(T)} |db_{r(\overline{x})}| + |u_{r(\overline{x})}|$.

*Proof.* We first establish the bound on $|\rho_n|$ during execution. Before execution, $\rho$ is a $T$-reduct of $db$. Hence, by Proposition 1, $|\rho_n| \leq \max_{r(\overline{x})} |db_{r(\overline{x})}| \leq M$ before execution starts. Now note that the only line that updates $\rho_n$ is line 19, executed while visiting node $n$ in the bottom-up traversal of $T$. This line is only applied once for every node $n$. Hence, since at the end of execution the collection of modified GMRs $\rho_m$ for $m \in N$ form a $T$-reduct of $db + u$, we know that after executing line 19, $\rho_n$ contains exactly the content for the $T$-reduct of $db + u$. Hence, by Proposition 1, $|\rho_n| \leq \max_{r(\overline{x})} |(db + u)_{r(\overline{x})}| \leq \max_{r(\overline{x})} |db_{r(\overline{x})}| + |u_{r(\overline{x})}| = M$.

We are now ready to establish the bounds on $|\Delta_n|$. Clearly, $|\Delta_n| \leq M$ during the initialization of $\Delta_n$ done in lines 14–17. Now consider that we are executing the bottom-up traversal of $T$ in lines 18–23 and that $n$ is the currently visited node. We have already established that both before and after applying the update $\Delta_n$ to $\rho_n$ we have 19 $|\rho_n| \leq M$. This implies that $|\Delta_n| \leq 2M$: in the worst case $\Delta_n$ deletes all existing tuples in $\rho_n$ and adds $M$ new ones. To see that $|\Delta_p| \leq 4M$ after executing line 21 we consider two cases. If $n$ is visited before $m$ in the bottom-up traversal of $T$, then $\Delta_p$ is necessarily empty before executing line 21 and hence $|\Delta_p| = |\pi_{var(p)}(\rho_m \ltimes_{pred(p)} \Delta_n)|$. Because, by definition of GJTs, $p$ has either $m$ or $n$ as a guard, it follows that every tuple in $\pi_{var(p)}(\rho_m \ltimes_{pred(p)} \Delta_n)$ is either a projected version of some tuple in $\rho_m$, or a projected version of some tuple in $\Delta_n$. As such, $|\Delta_p| \leq \max(|\rho_m|, |\Delta_n|) = 2M$. If, on the other hand, $m$ is visited before $n$ in the bottom-up traversal of $T$, then $\Delta_p$ necessarily contains the result computed during executing line 21 while visiting $m$. By the reasoning of the previous case, $|\Delta_p| \leq 2M$ before executing line 21 while visiting $n$. Furthermore, by the same reasoning we know that $|\pi_{var(p)}(\rho_m \ltimes_{pred(p)} \Delta_n)| \leq 2M$. Hence, at most $2M$ new tuples can be added to $\Delta_p$. As such, $|\Delta_p| \leq 4M$ after executing line 21 when visiting $n$. To see that also $|\Delta_p| \leq 4M$ after executing line 23 when visiting $n$, it suffices to observe that $\Delta_p$ starts out empty, and $|\Delta_n| \leq 2M$ (as already established). □

Combining Propositions 4 and 5 we obtain:

**Theorem 1** *Assume that all join indexes in the* $(T, N)$*-rep have access time* $g$*, and that all indexes (join and enumeration) have update time* $h$*, where* $g$ *and* $h$ *are monotone functions.* $\text{UPDATE}_{T,N}(\rho, u)$ *runs in time*

$$\mathcal{O}\left(|T| \cdot (4M + h(M, 4M) + g(M, 4M))\right).$$

*where* $M = \max_{r(\overline{x}) \in at(T)} |db_{r(\overline{x})}| + |u_{r(\overline{x})}|$.

### 4.4 IEDyn

GDyn provides a general framework for dynamic query processing in the presence of arbitrary $\theta$-joins. In this section we instantiate GDyn to the specific setting where queries mention only inequality predicates ($<, \leq, >, \geq$).[2] We refer to this instantiation as IEDyn. Concretely, IEDyn uses the following data structures for its enumeration and join indexes. Let $R$ be a GMR over $\overline{x}$, $\theta$ be a conjunction of inequalities, and $\overline{y}, \overline{z}$ be hyperedges such that $\overline{z} \subseteq \overline{x}$ or $\overline{z} \subseteq \overline{y}$. The data structure underlying the enumeration and join index of $R$ by $(\theta, \overline{y})$ resp. $(\theta, \overline{y}, \overline{z})$ depends on the number of inequalities in $\theta$.

**(1) No inequality.** In this case $\theta$ is hence equivalent to `true`, and the enumeration and join index hence only have to deal with equijoins. Concretely, the enumeration index of $R$ on $(\theta, \overline{y})$ is obtained by creating a normal (hash-based) index of $R$ on the variables that $\overline{x}$ and $\overline{y}$ have in common. Then, for any $\overline{y}$-tuple $\mathbf{t}$, $R \ltimes \mathbf{t}$ can be enumerated with constant delay by first using the hash index to find the corresponding $(\overline{x} \cap \overline{y})$-group of $R$, and enumerating the elements of that group.

To obtain the join index by $(\theta, \overline{y}, \overline{z})$ we discern two cases. If $\overline{z} \subseteq \overline{y}$, then the same index as for enumeration is re-used except that in addition, for each $(\overline{x} \cap \overline{y})$-group of $R$, we cache the sum of all multiplicities in that group. This allows to evaluate $\pi_{\overline{z}}(R \ltimes S)$ in time $\mathcal{O}(|S|)$ independently of $|R|$), as follows. Initialize an empty GMR to hold the join result. Group $S$ on the variables in $\overline{x} \cap \overline{y}$ in $O(|S|)$ time (using hashing). For each group in $S$, use the index on $R$ to locate the corresponding group of $R$ in $\mathcal{O}(1)$ time, and retrieve the cached sum-of-multiplicities $\mu$ of that group. Then iterate over the tuples $\mathbf{s}$ of the $S$-group one by one, and add $S(\mathbf{s}) \times \mu$ to the multiplicity of $\mathbf{s}[\overline{z}]$ in the result GMR. (If $\mathbf{s}[\overline{z}]$ has not been added to the result before, this multiplicity is zero.) Repeat this process for all tuples in the $S$-group, and for every group in $S$. Since we only scan $|S|$ once, the total time is $\mathcal{O}(|S|)$.

If $\overline{z} \subseteq \overline{x}$, no special data structure is required: we can compute $\pi_{\overline{z}}(R \ltimes S)$ in $\mathcal{O}(|R| + |S|)$ time by first computing $\pi_{\overline{x} \cap \overline{y}} S$ in $\mathcal{O}(|S|)$ time, and then repeating the above process with the roles of $R$ and $S$ reversed.

We conclude that in this case the enumeration time is $\mathcal{O}(1)$, the access time is $\mathcal{O}(|S|)$ if $\overline{z} \subseteq \overline{x}$ and $\mathcal{O}(|R| + |S|)$ otherwise, and the update time is $\mathcal{O}(|\Delta R|)$.

**(2) Single inequality.** Assume that $\theta = x > y$ with $x \in \overline{x}$ and $y \in \overline{y}$ (the other cases $x < y, x \leq y$, $x \geq y$ are similar). Then we build a hash-based index $I$ of $R$ on $\overline{x} \cap \overline{y}$, sorting each group in descending order on $x$.

[2] Note that such queries may also contain equijoins by sharing variables between atoms.

In Section 4.1 we have illustrated, by means of example, that this realizes an enumeration index by $(\theta, \overline{y})$ with constant delay. At the end of the same section we have also illustrated that, when $\overline{z} \subseteq \overline{x}$, this also realizes a join index of $R$ by $(\theta, \overline{y}, \overline{z})$ with access time $\mathcal{O}(|R| + |S| + |S| \log |S|) = \mathcal{O}(|R| + |S| \log |S|)$.[3] When $\overline{z} \subseteq \overline{x}$ the same procedure, but using $S$ for the outer loop, and $R$ for the inner loop, can be used to realize a join index of $R$ by $(\theta, \overline{y}, \overline{z})$ with access time $\mathcal{O}(|R| + |S| \log |S|)$.

Note that, because we need to keep data sorted the update time of these indexes is $\mathcal{O}\left(|\Delta R| \log(|R| + |\Delta R|)\right)$.

We conclude that in this case the enumeration time is $\mathcal{O}(1)$, the access time is $\mathcal{O}(|R| + |S| \log |S|)$, and the update time $\mathcal{O}\left(|\Delta R| \log(|R| + |\Delta R|)\right)$.

**(3) Multiple inequalities.** Assume that $\theta = x_1 > y_1 \wedge x_2 > y_2 \wedge \cdots \wedge x_k > y_k$ with $x_1, \ldots, x_k \in \overline{x}$ and $y_1, \ldots, y_k \in \overline{y}$. (The reasoning where some of the $>$ are replaced by $<$ is completely analogous.) Then, as with the case with single inequalities, we build a hash-based index $I$ of $R$ on $\overline{x} \cap \overline{y}$ but now sort each group lexicographically on $(x_1, \ldots, x_k)$ (each $x_i$ in descending order). In addition, for each group and each $i$ $(1 \leq i \leq k)$ we record the smallest $x_i$-value present in the group.

The fact that we have multiple inequalities complicates matters, in the sense that enumeration delay becomes logarithmic instead of constant. We can see this as follows. To enumerate $R \ltimes_\theta \mathbf{t}$ given $\overline{y}$-tuple $\mathbf{t}$ we first use $I$ to obtain a pointer to $R \ltimes \mathbf{t}[\overline{x} \cap \overline{y}]$ in $\mathcal{O}(1)$ time. Initialize $(m_1, \ldots, m_k)$ such that $m_i$ is the smallest $x_i$-value in the group. Then start enumerating $R \ltimes \mathbf{t}[\overline{x} \cap \overline{y}]$ with constant delay and in decreasing lexicographic order. Yield the current pair $(\mathbf{s}, \mu)$ that is being enumerated, provided that $\mathbf{s}[x_i] > \mathbf{t}[y_i]$ for all $i$. In contrast to the case where there is a single inequality, however, we cannot deduce that all subsequent $\mathbf{s}$ will fail $\theta$ we find that $\mathbf{s}[x_i] \leq \mathbf{t}[y_i]$ for some $i$. The solution then is to find the next tuple in the group that occurs after $\mathbf{s}$ in sorted order, but is larger than $\mathbf{t}$. Concretely, let $i$ be the smallest index such that $\mathbf{s}(x_i) \leq \mathbf{t}(y_i)$ yet $\mathbf{s}(x_j) > \mathbf{t}(x_j)$ for all $j < i$. Let $\mathbf{s'}$ be the tuple obtained from $\mathbf{s}$ by setting $\mathbf{s}(x_j) := m_j$ for all $j \geq i$. Then, using binary search, find the next tuple that lexicographically larger or equal than $\mathbf{s'}$, and re-continue enumeration from there. This binary search takes $\mathcal{O}(\log |R|)$ time, which causes the logarithmic delay.

Having multiple inequalities also complicates the update processing, since the sorted order no longer can be exploited to speed up join computation. In this case, therefore, we simply do a nested loop join per group, which yields a total access time of $\mathcal{O}(|R| \times |S|)$. Designing an join index with better access time is a interesting avenue for future work.

We conclude that in this case the enumeration time is $\mathcal{O}(\log |R|)$, the access time is $\mathcal{O}(|R| \times |S|)$, and the update time is $\mathcal{O}\left(|\Delta R| \log(|R| + |\Delta R|)\right)$.

**Complexity of** IEDyn**.** By plugging in the above-mentioned delay into Proposition 3 and the above-mentioned access time and update time into Corollary 1, we obtain the following complexity of IEDyn.

**Theorem 2** *Assume that $(T, N)$ is a plan in which all predicates are inequalities and let all enumeration and join indexes be as described above. Then* ENUM *enumerates with delay $\mathcal{O}(|N| \log(\max_{r(\overline{x})} |db_{r(\overline{x})}|))$ and* UPDATE *processes updates in time $\mathcal{O}(|T|M^2)$ where $M = \max_{r(\overline{x}) \in at(T)} |db_{r(\overline{x})}| + |u_{r(\overline{x})}|$.[4] If $T$ is such that each edge is labeled by at most one predicate, then the enumeration delay is $\mathcal{O}(|N|)$ and update time is $\mathcal{O}(|T|M \log M)$. If $T$ has no inequalities, the update time is $\mathcal{O}(|T|M)$.*

A *simple* GJT is a GJT without predicates where $var(p) \subseteq var(n)$ for every node $n$ with parent $p$. In a simple GJT every child is hence a guard of its parent. For simple GJTs, the update processing time is optimal, in the following sense.

**Theorem 3** *If $T$ is simple, then* UPDATE *processes updates in time $\mathcal{O}(|T| \max_{r(\overline{x})} |u_{r(\overline{x})}|)$, which is independent of $|db|$.*

*Proof.* Using the fact that every node is a guard of its parent, it is straightforward to prove by induction on the height of a node $n$ in $T$ (defined as the length of the shortest path from $n$ to a leaf in $T$) that $|\Delta\rho_n| \leq \max_{r(\overline{x})} |u_{r(\overline{x})}|$, for each $n$. Since $T$ does not contain any predicates, all join indexes that are created are as described in the paragraph "(1) No inequality" above. In particular, since every node is guard of its parent, we have that for every join index $S_m$ by $(pred(p), pred(n), var(p))$ that is created we have $var(p) \subseteq pred(m)$. For this particular case, the access time to execute the semi-join $\pi_{var(p)}(\rho_m \ltimes \Delta_n)$ is $\mathcal{O}(|\Delta|_n) = \mathcal{O}(\max_{r(\overline{x})} |u_{r(\overline{x})}|)$. By now plugging in this access time and the linear index update time into Proposition 4 the result follows. □

In [8], Berkholz, Keppeler, and Schweikardt show that, unless the Online Matrix-Vector Multiplication conjecture [24] is false, the class of conjunctive queries that allow both (1) constant-delay enumeration of query

[3] Strictly speaking, we described in Section that $R$ needs to be sorted lexicographically, first on $\overline{x} \cap \overline{y}$, and then on $x$. The grouping + sorting of the enumeration index obtains the same result.

[4] In the conference version of this paper [26] there was an incorrect claim: we stated that updates could be processed in time $O(M \cdot \log(M))$ in the general case of multiple inequalities. We then found a bug in our proof and we currently do not know if this bound can be achieved.

results (in data complexity) and (2) update processing time that is linear in $|u|$ (again in data complexity) for every update $u$, is exactly the class of so-called *q-hierarchical queries*. While we forego a formal definition of this class, we show in [25] that a CQ $Q$ is $q$-hierarchical if, and only if, there exists a plan $(T, N)$ for $Q$ such that $T$ is simple. Since, by the results above, GDyn, has both constant-delay enumeration and update time $\mathcal{O}(|u|)$ (in data complexity) for exactly these queries, GDyn hence meets the theoretical lower bound.

## 5 Computing Query Plans

We say that $(T, N)$ is a *plan for* GCQ $Q$, or that $Q$ has plan $(T, N)$, if $Q$ and $\mathcal{Q}[\![T, N]\!]$ are the same query, up to reordering of atoms and predicates, i.e., if $\#at(Q) = \#at(T)$, $pred(Q) = pred(T)$, and $out(Q) = var(N)$. Here, $\#at(X)$ denotes the multi-set of atoms occurring in object $X$.

By the results of Section 4, it follows that we can dynamically process a given GCQ $Q$ by first computing a plan for $Q$, and subsequently applying GDyn on that plan. In this section we show *how* to compute a plan for $Q$ by describing two algorithms.

1. The first algorithm computes a *GJT pair* for $Q$. Here, a *GJT pair* is a pair $(T', N')$ defined exactly like a query plan, except that $T'$ need not be binary, and $N'$ need not be sibling-closed. A query plan is hence a particular kind of GJT pair. We call $(T', N')$ a *GJT pair for* $Q$ if $\#at(Q) = \#at(T')$, $pred(Q) = pred(T')$, and $out(Q) = var(N')$.
2. The second algorithm transforms this GJT pair into an equivalent query plan. Here, two GJT pairs $(T, N)$ and $(T', N')$ are *equivalent* if $\#at(T) = \#at(T')$, $pred(T) = pred(T')$, and $var(N) = var(N')$.

Clearly, the plan resulting from the composition of the two algorithms must be a plan for $Q$.

Before describing these two algorithms, it is important to emphasize that there are GCQs for which no GJT pair exists (and, consequently, for which no query plans exists). We give some examples in Example 2 below. In particular, for full conjunctive queries (i.e., GCQs without $\theta$-joins and projections), the results of [25] imply that a GJT pair exists for a full CQ $Q$ if, and only if, $Q$ is *acyclic*, a well-studied class of queries [1, 47]. Similarly, the results imply that for conjunctive queries (with projections, but still without $\theta$-joins) a GJT pair exists if an only if the query is *free-connex acyclic*, another well-studied class [6]. The existing definitions of acyclicity and free-connex acyclicity are given for CQs only. Given the previous discussion, we extend these notions to GCQs as follows.

**Definition 6** A GCQ $Q$ is *free-connex acyclic* if it has a GJT pair. It is *acyclic* if $full(Q)$ has a GJT pair. A GCQ that is not acyclic is *cyclic*.

In particular, every full GCQ that is acylic is also free-connex acyclic. Also note that, since $out(full(Q)) = var(Q)$, a GJT pair will exist for $full(Q)$ if an only if there exists a GJT $T$ for $Q$, i.e., a GJT with $\#at(Q) = \#at(T)$ and $pred(Q) = pred(T)$. Indeed, if $T$ is a GCQ for $Q$ then $(T, N)$ with $N$ the set of all nodes in $T$, is a GJT pair for $Q$: clearly $N$ is connex and $var(N) = var(T) = out(Q)$. For this reason, free-connex acyclicity is a stronger requirement than acyclicity: acyclicity only requires that a GJT for $Q$ exists while free-connex acyclicity requires in addition that there exists a connex subset with $out(Q) = var(N)$.

*Example 2* The trees $T_1$ and $T_2$ depicted in Fig. 4 are GJTs for the full GCQs

$$Q_1 = \big(r(x, y) \bowtie s(y, z) \bowtie t(z, v) \mid y < v\big), \text{ and}$$
$$Q_2 = \big(r(x, y) \bowtie s(y, z, w) \bowtie t(u, v) \mid x < z \wedge w < u\big),$$

respectively. These queries are hence acyclic. In contrast, $r(x, y) \bowtie s(y, z) \bowtie t(x, z)$ (also known as the triangle query) is the prototypical cyclic join query.

Let $Q_2' = \pi_{y,z,w,u}(Q_2)$. $Q_2'$ is free-connex acyclic since the pair $(T_2, N_2)$ of Fig. 4 is a GJT pair for $Q_2$. By contrast, there is no GJT pair for $Q_1' = \pi_{y,z}(Q_1)$ that contains tree $T_1$. Indeed, observe that any connex set of $T_1$ must include the root, which includes $x \notin out(Q_1')$. Finally, it can be verified that there is no GJT pair for $\pi_{x,v}(Q_1)$; this query is hence not free-connex acyclic.

Given that plans only exist for free-connex acyclic queries, it is desirable to be able to check free-connex acyclicity. In this respect, we develop an algorithm in Section 5.1 that checks whether $Q$ is acyclic and free-connex acyclicity, and if so, computes a GJT pair for $Q$. This algorithm hence realizes step (1) above. Subsequently, step (2) above is realized in Section 5.2 where we discuss how to transform GJT pairs into equivalent query plans.

### 5.1 Computing GJT pairs

The canonical algorithm for checking acyclicity of normal conjunctive queries is the GYO algorithm [1]. The algorithm described in this section is a generalisation of the GYO algorithm that checks free-connex acyclicity in addition to normal acyclicity and deals with GCQs featuring $\theta$-join predicates instead of CQs that have equality joins only. We first recall the classical GYO algorithm and then formulate its extension.

*5.1.1 Classical GYO*

The GYO algorithm operates on *hypergraphs*. A *hypergraph* $H$ is a set of non-empty hyperedges. Recall from Section 3 that a hyperedge is just a finite set of variables. Every GCQ is associated to a hypergraph as follows.

**Definition 7** Let $Q$ be a GCQ. The hypergraph of $Q$, denoted $\mathit{hyp}(Q)$, is the hypergraph

$$\mathit{hyp}(Q) = \{\overline{x} \mid r(\overline{x}) \text{ is an atom of } Q \text{ with } \overline{x} \neq \emptyset\}.$$

The GYO algorithm checks acyclicity of a normal conjunctive query $Q$ by constructing $\mathit{hyp}(Q)$ and repeatedly removing *ears* from this hypergraph. If ears can be removed until only the empty hypergraph remains, then the query is acyclic; otherwise it is cyclic.

An *ear* in a hypergraph $H$ is a hyperedge $e$ for which we can divide its variables into two groups: (1) those that appear exclusively in $e$, and (2) those that are contained in another hyperedge $\ell$ of $H$. A variable that appears exclusively in a single hyperedge is also called an *isolated variable*. Thus, ear removal corresponds to executing the following two reduction operations.

- Remove isolated variables: select a hyperedge $e$ in $H$ and remove isolated variables from it; if $e$ becomes empty, remove $e$ it altogether from $H$.
- Subset elimination: remove hyperedge $e$ from $H$ if there exists another hyperedge $\ell$ for which $e \subseteq \ell$.

The *GYO reduction* of a hypergraph is the hypergraph that is obtained by executing these operations until no further operation is applicable. The following result is standard; see e.g., [1] for a proof.

**Proposition 6** *A CQ $Q$ is acyclic if and only if the GYO-reduction of $\mathit{hyp}(Q)$ is the empty hypergraph.*

*5.1.2 GYO-reduction for GCQs*

In order to extend the GYO-reduction to check free-connex acyclicity (not simply acyclicity) of GCQs (not simply standard CQs), we will: (1) Redefine the notion of being an ear to take into account the predicates; and (2) transform the GYO-reduction into a two-stage procedure. The first stage allows to check that a connex set with exactly $\mathit{out}(Q)$ can exist while the first and second stage combined check that the query is acyclic.

Our algorithm operates on *hypergraph triplets* instead of hypergraphs, which are defined as follows.

**Definition 8** A *hypergraph triplet* is a triple

$$\mathcal{H} = (\mathit{hyp}(\mathcal{H}), \mathit{out}(\mathcal{H}), \mathit{pred}(\mathcal{H}))$$

with $\mathit{hyp}(\mathcal{H})$ a hypergraph, $\mathit{out}(\mathcal{H})$ a hyperedge, and $\mathit{pred}(\mathcal{H})$ a set of predicates.

Intuitively, the variables in $\mathit{out}(\mathcal{H})$ will correspond to the output variables of a query and the set $\mathit{pred}(\mathcal{H})$ will contain predicates that need to be taken into account when removing ears. Every GCQ is therefore naturally associated to a hypergraph triplet as follows.

**Definition 9** The hypergraph triplet of a GCQ $Q$, denoted $\mathcal{H}(Q)$, is the triplet $(\mathit{hyp}(Q), \mathit{out}(Q), \mathit{pred}(Q))$.

In order to extend the notion of an ear, we require the following definitions. Let $\mathcal{H}$ be a hypergraph triplet. Variables that occur in $\mathit{out}(\mathcal{H})$ or in at least two hyperedges in $\mathit{hyp}(\mathcal{H})$ are called *equijoin variables* of $\mathcal{H}$. We denote the set of all equijoin variables of $\mathcal{H}$ by $\mathit{jv}(\mathcal{H})$ and abbreviate $\mathit{jv}_{\mathcal{H}}(e) = e \cap \mathit{jv}(\mathcal{H})$. A variable $x$ is *isolated* in $\mathcal{H}$ if it is not an equijoin variable and is not mentioned in any predicate, i.e., if $x \notin \mathit{jv}(\mathcal{H})$ and $x \notin \mathit{var}(\mathit{pred}(\mathcal{H}))$. We denote the set of isolated variables of $\mathcal{H}$ by $\mathit{isol}(\mathcal{H})$ and abbreviate $\mathit{isol}_{\mathcal{H}}(e) = e \cap \mathit{isol}(\mathcal{H})$. The *extended variables* of hyperedge $e$ in $\mathcal{H}$, denoted $\mathit{ext}_{\mathcal{H}}(e)$ is the set of all variables of predicates that mention some variable in $e$, except the variables in $e$ themselves:

$$\mathit{ext}_{\mathcal{H}}(e) = \bigcup\{\mathit{var}(\theta) \mid \theta \in \mathit{pred}(\mathcal{H}), \mathit{var}(\theta) \cap e \neq \emptyset\} \setminus e.$$

Finally, a hyperedge $e$ is a *conditional subset* of hyperedge $\ell$ w.r.t. $\mathcal{H}$, denoted $e \sqsubseteq_{\mathcal{H}} \ell$, if $\mathit{jv}_{\mathcal{H}}(e) \subseteq \ell$ and $\mathit{ext}_{\mathcal{H}}(e \setminus \ell) \subseteq \ell$. We omit subscripts from our notation if the triplet is clear from the context.

*Example 3* In Fig. 5 we depict several hypergraph triplets. There, hyperedges in $\mathcal{H}$ are depicted by colored regions and variables in $\mathit{out}(\mathcal{H})$ are underlined. We use dashed lines to connect variables that appear together in a predicate. So, in $\mathcal{H}_1$, we have predicates $\theta_1, \theta_2$ with $\mathit{var}(\theta_1) = \{t, v\}$ and $\mathit{var}(\theta_2) = \{x, y\}$. Now consider triplet $\mathcal{H}_1$ in particular. It is the hypergraph triplet $\mathcal{H}(Q)$ for the following GCQ $Q$:

$$Q = \pi_{t,u,z,w}(r_1(s,t,u) \bowtie r_2(t,u) \bowtie r_3(u,w,x) \bowtie r_4(s,v) \bowtie r_5(w,z,y) \mid t < v \wedge x < y).$$

Moreover, $\mathit{jv}(\mathcal{H}_1) = \{s,t,u,w,z\}$ and $\mathit{isol}(\mathcal{H}_1) = \emptyset$. Furthermore, $\mathit{ext}_{\mathcal{H}_1}(\{v\}) = \{t\}$ since $\theta_1 = t < v$ shares variables with $\{v\}$. Finally $\mathit{jv}_{\mathcal{H}_1}(\{s,v\}) = \{s\} \subseteq \{s,t,u\}$ and $\mathit{ext}_{\mathcal{H}_1}(\{s,v\} \setminus \{s,t,u\}) = \mathit{ext}_{\mathcal{H}_1}(\{v\}) = \{t\} \subseteq \{s,t,u\}$. Therefore, $\{s,v\} \sqsubseteq_{\mathcal{H}_1} \{s,t,u\}$. Similarly, $\{t,u\} \sqsubseteq_{\mathcal{H}_1} \{s,t,u\}$.

We define ears in our context as follows.

**Definition 10** A hyperedge $e$ is an ear in a hypergraph triplet $\mathcal{H}$ if $e \in \mathit{hyp}(\mathcal{H})$ and either

1. we can divide its variables into two: (a) those that are isolated and (b) those that form a conditional subset of another hyperedge $\ell \in \mathit{hyp}(\mathcal{H}) \setminus \{e\}$; or

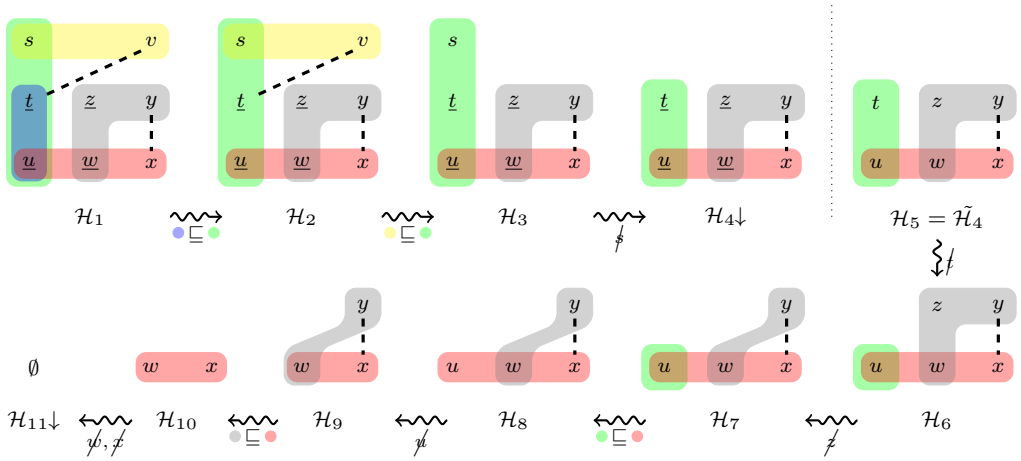


**Fig. 5** Illustration of GYO-reduction for GCQs. Colored regions depict hyperedges. Variables in *out* are underlined. Variables occurring in the same predicate are connected by dashed lines.

2. $e$ consists only of non-join variables, i.e., $jv(e) = \emptyset$ and $ext(e) = \emptyset$.

Note that case (2) allows for $\theta \in pred(\mathcal{H})$ with $var(\theta) \subseteq e$. We call predicates that are covered by a hyperedge in this sense *filters* because they correspond to filtering a single GMR instead of $\theta$-joining two GMRs. If, in case (2), there is no filter $\theta$ with $var(\theta) \subseteq e$, then $e = isol_{\mathcal{H}}(e)$. Similar to the classical GYO reduction, we can view ear removal as a rewriting process on triplets, where we consider the following reduction operations.

- (ISO) Remove isolated variables: select a hyperedge $e \in hyp(\mathcal{H})$ and remove a non-empty set $X \subseteq isol_{\mathcal{H}}(e)$ from it. If $e$ becomes empty, remove it from $hyp(\mathcal{H})$.
- (CSE) Conditional subset elimination: remove hyperedge $e$ from $hyp(\mathcal{H})$ if it is a conditional subset of another hyperedge $f$ in $hyp(\mathcal{H})$. Also update $pred(\mathcal{H})$ by removing all predicates $\theta$ with $var(\theta) \cap (e \setminus f) \neq \emptyset$.
- (FLT) Filter elimination: select $e \in hyp(\mathcal{H})$ and a non-empty subset of predicates $\Theta \subseteq pred(\mathcal{H})$ with $var(\Theta) \subseteq e$. Remove all predicates in $\Theta$ from $pred(\mathcal{H})$.

We write $\mathcal{H} \rightsquigarrow \mathcal{I}$ to denote that triplet $\mathcal{I}$ is obtained from triplet $\mathcal{H}$ by applying a single such operation, and $\mathcal{H} \rightsquigarrow^* \mathcal{I}$ to denote that $\mathcal{I}$ is obtained by a sequence of zero or more of such operations.

*Example 4* For the hypergraph triplets illustrated in Fig. 5 we have $\mathcal{H}_1 \rightsquigarrow \mathcal{H}_2 \rightsquigarrow \mathcal{H}_3 \rightsquigarrow \mathcal{H}_4$ and $\mathcal{H}_5 \rightsquigarrow \mathcal{H}_6 \rightsquigarrow \mathcal{H}_7 \rightsquigarrow \mathcal{H}_8 \rightsquigarrow \mathcal{H}_9 \rightsquigarrow \mathcal{H}_{10} \rightsquigarrow \mathcal{H}_{11}$. For each reduction, it is illustrated in the figure which set of isolated variables is removed, or which conditional subset is removed.

We write $\mathcal{H}\downarrow$ to denote $\mathcal{H}$ is in *normal form*, i.e., that no operation is applicable on triplet $\mathcal{H}$. Note that, because each operation removes at least one variable, hyperedge, or predicate, we will always reach a normal form after a finite number of operations. Furthermore, while multiple different reduction steps may be applicable on a given triplet $\mathcal{H}$, the order in which we apply them does not matter:

**Proposition 7 (Confluence)** *Whenever $\mathcal{H} \rightsquigarrow^* \mathcal{I}_1$ and $\mathcal{H} \rightsquigarrow^* \mathcal{I}_2$, there exists $\mathcal{J}$ such that $\mathcal{I}_1 \rightsquigarrow^* \mathcal{J}$ and $\mathcal{I}_2 \rightsquigarrow^* \mathcal{J}$.*

Because the proof is technical but not overly enlightning, we defer it to Appendix B.1. A direct consequence is that normal forms are unique: if $\mathcal{H} \rightsquigarrow^* \mathcal{I}_1\downarrow$ and $\mathcal{H} \rightsquigarrow^* \mathcal{I}_2\downarrow$ then $\mathcal{I}_1 = \mathcal{I}_2$.

Let $\mathcal{H}$ be a triplet. The residual of $\mathcal{H}$, denoted $\tilde{\mathcal{H}}$, is the triplet $(hyp(\mathcal{H}), \emptyset, pred(\mathcal{H}))$, i.e., the triplet where $out(\mathcal{H})$ is set to $\emptyset$. A triplet is *empty* if it equals $(\emptyset, \emptyset, \emptyset)$.

Our main result in this section states that to check whether a GCQ $Q$ is free-connex acyclic it suffices to start from $\mathcal{H}(Q)$ and do a two stage reduction: the first from $\mathcal{H}(Q)$ until a normal form $\mathcal{I}\downarrow$ is reached, and the second from the residual of $\mathcal{I}\downarrow$, until another normal form $\mathcal{J}$ is reached.[5]

**Theorem 4** *Let $Q$ be a GCQ. Assume $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}\downarrow$ and $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}\downarrow$. Then the following hold.*

1. *$Q$ is acyclic if, and only if, $\mathcal{J}$ is the empty triplet.*
2. *$Q$ is free-connex acyclic if, and only if, $\mathcal{J}$ is the empty triplet and $var(hyp(\mathcal{I})) = out(Q)$.*
3. *For every GJT $T$ of $Q$ and every connex subset $N$ of $T$ it holds that $var(hyp(\mathcal{I})) \subseteq var(N)$.*

We devote Section 5.1.3 to the proof.

*Example 5* Fig. 5 illustrates the two-stage sequence of reductions starting from $\mathcal{H}(Q)$ with $Q$ the GCQ of

[5] Note that because we set $out(\mathcal{I}) = \emptyset$ on the residual, new variables may become isolated and therefore more reductions steps may be possible on the normal form of $\mathcal{I}$.

Example 3. Note that $\mathcal{H}(Q) = \mathcal{H}_1$ and $\mathcal{H}_5$ is the residual of $\mathcal{H}_4$. Because we end with the empty triplet, $Q$ is acyclic but not free-connex since $out(Q) \subsetneq var(\mathcal{H}_4)$.

Theorem 4 gives us a decision procedure for checking free-connex acyclicity of GCQ $Q$. From its proof in Section 5.1.3, we can actually derive an algorithm for constructing a GJT pair for $Q$. At its essence, this algorithm starts with the set of atoms appearing in $Q$, and subsequently uses the sequence of reduction steps from Theorem 4 to construct a GJT from it, at the same time checking free-connex acyclicity. Every reduction step causes new nodes to be added to the partial GJT constructed so far. We will refer to such partial GJTs as *Generalized Join Forests* (GJF).

**Definition 11 (GJF)** A *Generalized Join Forest* is a set $F$ of pairwise disjoint GJTs s.t. for distinct trees $T_1, T_2 \in F$ we have $var(T_1) \cap var(T_2) = var(n_1) \cap var(n_2)$ where $n_1$ and $n_2$ are the roots of $T_1$ and $T_2$.

Every GJF encodes a hypergraph as follows.

**Definition 12** The hypergraph $hyp(F)$ associated to GJF $F$ is the hypergraph that has one hyperedge for every non-empty root node in $F$,

$$hyp(F) = \{var(n) \mid n \text{ root node in } F, var(n) \neq \emptyset\}.$$

The GJT construction algorithm does not manipulate hypergraph triplets directly. Instead, it manipulates *GJF triplets*. A GJF triplet is defined like a hypergraph triplet, except that it has a GJF instead of a hypergraph.

**Definition 13** A *GJF triplet* is a triple $\mathbb{F} = (forest(\mathbb{F}), out(\mathbb{F}), \Theta_{\mathbb{F}})$ with $forest(\mathbb{F})$ a GJF, $out(\mathbb{F})$ a hyperedge, and $\Theta_{\mathbb{F}}$ a set of predicates. Every GJF triplet $\mathbb{F}$ induces a hypergraph triplet $\mathcal{H}(\mathbb{F}) = (hyp(forest(\mathbb{F})), out(\mathbb{F}), \Theta_{\mathbb{F}})$.

The algorithm for constructing a GJT pair for a given GCQ $Q$ is now shown in Algorithm 2. It starts in line 2 by initializing the GJF triplet $\mathbb{F}$ to $\mathbb{F} = (forest(Q), out(Q), pred(Q))$. Here, $forest(Q)$ is the GJF obtained by creating, for every atom $r(\overline{x})$ that occurs $k > 0$ times in $Q$, $k$ corresponding leaf nodes labeled by $r(\overline{x})$. In Lines 3–4, Algorithm 2 then performs the first phase of reduction steps of Theorem 4. To this end, it checks whether a reduction operation is applicable to $\mathcal{H}(\mathbb{F})$ and, if so, *enacts* this operation by modifying $\mathbb{F}$ as follows.

- (ISO). If the reduction operation on the hypergraph triplet $\mathcal{H}(\mathbb{F})$ were to remove a non-empty subset $X$ of isolated variables from hyperedge $e$, then $\mathbb{F}$ is modified as follows. Let $n_1, \ldots, n_k$ be all the root nodes in $forest(\mathbb{F})$ that are labeled by $e$. Merge the corresponding trees into one tree by creating a new node $n$ with $var(n) = e$ and attaching $n_1, \ldots, n_k$ as children to it with $pred(n \to n_i) = \emptyset$ for $1 \leq i \leq k$. Then, enact the removal of $X$ by creating a new node $p$ with $var(p) = e \setminus X$ and attaching $n$ as child to it with $pred(p \to n) = \emptyset$.
- (CSE) If the reduction operation on $\mathcal{H}(\mathbb{F})$ were to remove a hyperedge $e$ because it is a conditional subset of another hyperedge $\ell$, then $\mathbb{F}$ is modified as follows. Let $n_1, \ldots, n_k$ (resp. $m_1, \ldots, m_l$) be all the root nodes in $forest(\mathbb{F})$ that are labeled by $e$ (resp. $\ell$), and let $T_1, \ldots, T_k$ (resp. $U_1, \ldots, U_l$) be their corresponding trees. Similar to the previous case, merge the $T_i$ (resp. $U_j$) into a single tree with new root $n$ labeled by $e$ (resp. $m$ labeled by $\ell$). Then enact the removal of $e$ by creating a new node $p$ with $var(p) = \ell$ and attaching $n$ and $m$ as children with $pred(p \to n) = \{\theta \in pred(\mathbb{F}) \mid var(\theta) \cap (e \setminus \ell) \neq \emptyset\}$ and $pred(p \to m) = \emptyset$.
- (FLT) If the reduction operation on $\mathcal{H}(\mathbb{F})$ were to remove non-empty set of predicates $\Theta$ because there exists a hyperedge $e$ with $var(\Theta) \subseteq e$, then $\mathbb{F}$ is modified as follows. Let $n_1, \ldots, n_k$ be all the root nodes in $forest(\mathbb{F})$ that are labeled by $e$. Merge the corresponding trees into one tree by creating a new root $n$ labeled by $e$, and attaching $n_1, \ldots, n_k$ as children with $pred(n \to n_i) = \Theta$. Enact the removal of $\Theta$ by removing all $\theta \in \Theta$ from $\Theta(\mathbb{F})$.

**Algorithm 2** Compute a GJT pair

```
1: Input: A GCQ Q.
2: 𝔽 ← (forest(Q), out(Q), pred(Q))
3: while a reduction step is applicable to ℋ(𝔽) do
4:    enact the reduction on 𝔽
5: X ← set of all root nodes in 𝔽
6: set pred(𝔽) := ∅
7: while a reduction step is applicable to ℋ(𝔽) do
8:    enact the reduction on 𝔽
9: if ℋ(𝔽) is not the empty triplet then
10:    error "Q is not acyclic"
11: else
12:    T ← tree obtained by connecting all root nodes of 𝔽's
       forest to a new root, labeled by ∅
13:    N ← all nodes in X and their ancestors in T
14:    return (T, N)
```

It is straightforward to check that these modifications of the forest triplet $\mathbb{F}$ faithfully enact the corresponding operations on $\mathcal{H}(\mathbb{F})$, in the following sense.

**Lemma 4** *Let $\mathbb{F}$ be a forest triplet and assume $\mathcal{H}(\mathbb{F}) \rightsquigarrow \mathcal{I}$. Let $\mathbb{G}$ be the result of enacting this reduction operation on $\mathbb{F}$. Then $\mathbb{G}$ is a valid forest triplet and $\mathcal{H}(\mathbb{G}) = \mathcal{I}$.*

We continue the explanation of Algorithm 2. In line 5, Algorithm 2 records the set of root nodes obtained after the first stage of reductions. It then sets $out(\mathbb{F}) = \emptyset$ in line 6 and continues with the second stage of reductions in lines 7–8. It then employs Theorem 4 to check

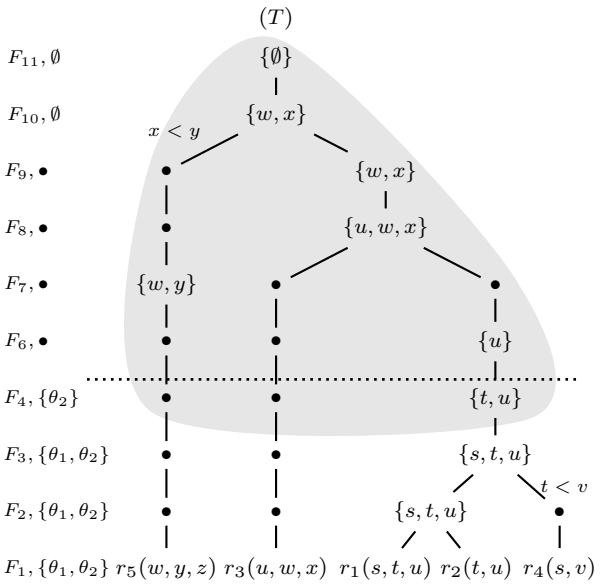


**Fig. 6** GJT Construction by GYO-reduction.

acyclicity of $Q$. If $Q$ is not acyclic, it reports this in lines 9–10. If $Q$ is acyclic, then we know by Theorem 4 that $\mathcal{H}(\mathbb{F})$ has become the empty triplet. Note that $\mathcal{H}(\mathbb{F})$ can be empty only if all the roots of $\mathbb{F}$'s join forest are labeled by the empty set of variables. As such, we can transform this forest into a join tree $T$ by linking all of these roots to a new unique root, also labeled $\emptyset$. This is done in line 12. In line 13, the set of nodes $N$ is computed, and consists of all nodes identified at the end of the first stage (line 5) plus all of their parents in $T$.

We will prove in Section 5.1.3 that Algorithm 2 is correct, in the following sense.

**Theorem 5** *Given a GCQ $Q$, Algorithm 2 reports an error if $Q$ is cyclic. Otherwise, it returns a GJT pair $(T, N)$ with $T$ a GJT for $Q$. If $Q$ is free-connex acyclic, then $(T, N)$ is GJT pair for $Q$. Otherwise, $out(Q) \subsetneq var(N)$, but $var(N)$ is minimal in the sense that for every other GJT pair $(T', N')$ with $T'$ a GJT for $Q$ we have $var(N) \subseteq var(N')$.*

It is straightforward to check that this algorithm runs in polynomial time in the size of $Q$.

*Example 6* In Fig. 6, we show a GJT $T$ and use this *GJT* to illustrate a number of GJFs $F_1, \ldots, F_{10}$ in the following way: let level 1 be the leaf nodes, level 2 the parents of the leaves, and so on. Then we take GJF $F_i$ to be the set of all trees rooted at nodes at level $i$, for $1 \leq i \leq 10$, and with each level $i$, we mention the set of remaining predicates $\theta_i$ for $1 \leq i \leq k$ where $k$ is the number of predicates in $Q$. Nodes (resp. predicates with each $F_i$) labeled by "$\bullet$" in Fig. 6 indicates that the node (and hence tree, resp. predicates) was already present in $F_{i-1}$ and did not change. These should hence not be interpreted as new nodes (resp. predicates changed). With this coding of forests, it is easy to see that for all $1 \leq i \leq 9$, $F_i = hyp(\mathcal{H}_i)$ with $\mathcal{H}_i$ illustrated in Fig. 5 (note here that the hypergraph of residual of $\mathcal{H}_4$ i.e. $\mathcal{H}_5$ is the same as $\mathcal{H}_4$, hence we do not show the corresponding $F_5$). Furthermore, $pred(F_i) = pred(Q) \setminus pred(\mathcal{H}_i)$ with $Q$ the GCQ from Example 3. As such, the tree illustrates the sequence of GJF triplets that is obtained by enacting the hypergraph reductions illustrated in Fig. 5. For example, let $\mathbb{F}_1 = (F_1, out(Q), pred(Q)$. After enacting the removal of hyperedge $\{t, u\}$ from $\mathcal{H}_1$ to obtain $\mathcal{H}_2$ we obtain $\mathbb{F}_2 = (F_2, out(Q), pred(Q))$. Here, $F_2$ is obtained by merging the single-node trees (i.e. labelled by the atoms in $Q$) $\{s, t, u\}$ and $\{t, u\}$ in to a single tree with root $\{s, t, u\}$. The shaded area illustrate the nodes in the connex subset $N$ computed by Algorithm 2.

We stress that Algorithm 2 is non-deterministic in the sense that the pair $(T, N)$ returned depends on the order in which the reduction operations are performed.

### *5.1.3 Correctness*

To prove theorems 4 and 5 we show some propositions.

**Proposition 8** *Let $Q$ be a GCQ. Assume $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}\!\downarrow$ and $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}\!\downarrow$. If $\mathcal{J}$ is the empty triplet, then, when run on $Q$, Algorithm 2 returns a pair $(T, N)$ s.t. $T$ is a GJT for $Q$ and $var(N) = var(hyp(\mathcal{I}))$.*

*Proof.* Assume that $\mathcal{J}$ is the empty triplet. Algorithm 2 starts in line 3 by initializing $\mathbb{F} = (forest(Q), out(Q), pred(Q))$. Clearly, $\mathcal{H}(\mathbb{F}) = \mathcal{H}(Q)$ at this point. Algorithm 2 subsequently modifies $\mathbb{F}$ throughout its execution. Let $\mathbb{H}$ denote the initial version of $\mathbb{F}$; let $\mathbb{I}$ denote the version of $\mathbb{F}$ when executing line 5; let $\tilde{\mathbb{I}}$ denote the version of $\mathbb{F}$ after executing line 6 and let $\mathbb{J}$ denote the version of $\mathbb{F}$ when executing line 9. By repeated application of Lemma 4 we know that $\mathcal{H}(Q) = \mathcal{H}(\mathbb{H}) \rightsquigarrow^* \mathcal{H}(\mathbb{I})$. Furthermore, $\mathcal{H}(\mathbb{I})$ is in normal form. Since also $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}\!\downarrow$ and normal forms are unique, $\mathcal{H}(\mathbb{I}) = \mathcal{I}$. Therefore, $\mathcal{H}(\tilde{\mathbb{I}}) = \tilde{\mathcal{I}}$. Again by repeated application of Lemma 4 we know that $\tilde{\mathcal{I}} = \mathcal{H}(\tilde{\mathbb{I}}) \rightsquigarrow^* \mathcal{H}(\mathbb{J})$. Moreover, $\mathcal{H}(\mathbb{J})$ is in normal form. Since also $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}\!\downarrow$ and normal forms are unique, $\mathcal{H}(\mathbb{J}) = \mathcal{J}$. As $\mathcal{J}$ is empty, we will execute lines 12–14. Since $\mathcal{J}$ is the empty hypergraph triplet, every root of every tree in $forest(\mathbb{J})$ must be labeled by $\emptyset$. By definition of join forests, no two distinct trees in $forest(\mathbb{J})$ hence share variables. As such, the tree $T$ obtained in line 12 by linking all of these roots to a new unique root, also labeled $\emptyset$, is a valid GJT.

We claim that $T$ is a GJT for $Q$. Indeed, observe that $at(T) = at(Q)$ and the number of times that an atom

occurs in $Q$ equals the number of times that it occurs as a label in $T$. This is because initially $\mathit{forest}(\mathbb{H}) = \mathit{forest}(Q)$ and by enacting reduction steps we never remove nor add nodes labeled by atoms. Furthermore $\mathit{pred}(T) = \mathit{pred}(Q)$. This is because initially $\mathit{pred}(\mathbb{H}) = \mathit{pred}(Q)$ yet $\Theta_{\mathbb{J}}$ is empty. This means that, for every $\theta \in \mathit{pred}(Q)$, there was some reduction step that removed $\theta$ from the set of predicates of the current GJF triplet $\mathbb{F}$. However, when enacting reduction steps we only remove predicates after we have added them to $\mathit{forest}(\mathbb{F})$. Therefore, every predicate in $\mathit{pred}(Q)$ must occur in $T$. Conversely, during enactment of reduction steps we never add predicates to $\mathit{forest}(\mathbb{F})$ that are not in $\Theta_{\mathbb{F}}$, so all predicates in $T$ are also in $\mathit{pred}(Q)$. Thus, $T$ is a GJT for $Q$.

It remains to show that $N$ is a connex subset of $T$ and $\mathit{var}(N) = \mathit{var}(\mathit{hyp}(\mathcal{I}))$. To this end, let $X$ be the set of all root nodes of $\mathit{forest}(\mathbb{I})$, as computed in Line 5. Since $\mathbb{J}$ is obtained from $\tilde{\mathbb{I}}$ by a sequence of reduction enactments, and since such enactments only add new nodes and never delete them, $X$ is a subset of nodes of $\mathit{forest}(\mathbb{J})$ and therefore also of $T$. As computed in Line 13, $N$ consists of $X$ and all ancestors of nodes of $X$ in $T$. Then $N$ is a connex subset of $T$ by definition. Furthermore, since $\mathcal{H}(\mathbb{I}) = \mathcal{I}$, $\mathit{hyp}(\mathit{forest}(\mathbb{I})) = \mathit{hyp}(\mathcal{I})$. Thus, $\mathit{var}(X) = \mathit{var}(\mathit{hyp}(\mathbb{I})) = \mathit{var}(\mathit{hyp}(\mathcal{I}))$. Hence, to establish that $\mathit{var}(N) = \mathit{var}(\mathit{hyp}(\mathcal{I}))$ it suffices to show that $\mathit{var}(X) = \mathit{var}(N)$. Since $X \subseteq N$ the inclusion $\mathit{var}(X) \subseteq \mathit{var}(N)$ is immediate. To also establish $\mathit{var}(N) \subseteq \mathit{var}(X)$, let $n$ be an arbitrary but fixed node in $N$. If $n \in X$ then clearly $\mathit{var}(n) \subseteq \mathit{var}(X)$. If $n \notin X$ then $n$ was created during the sequence of reduction enactments in that transform $\tilde{\mathbb{I}}$ into $\mathbb{J}$. Now note that, whenever a new node $m$ is created during a reduction enactment on a GJF $\mathbb{G}$, there exists a root node of $\mathit{forest}(\mathbb{G})$ that contains all variables of $m$. From this observation and the fact that $n$ was created during a sequence of reduction enactments that start from $\tilde{\mathbb{I}}$, it follows that there there is some root node $r$ in $\tilde{\mathbb{I}}$ with $\mathit{var}(n) \subseteq \mathit{var}(r)$. Then, because $X$ contains all root nodes of $\tilde{\mathbb{I}}$, also $\mathit{var}(n) \subseteq \mathit{var}(X)$. Therefore, $\mathit{var}(N) = \mathit{var}(X) = \mathit{var}(\mathit{hyp}(\mathcal{I}))$. □

**Corollary 1 (Soundness)** *Let $Q$ be a GCQ and assume that $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}\!\downarrow$ and $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}\!\downarrow$. Then:*

1. *If $\mathcal{J}$ is the empty triplet then $Q$ is acyclic.*
2. *If $\mathcal{J}$ is the empty triplet and $\mathit{var}(\mathit{hyp}(\mathcal{I})) = \mathit{out}(Q)$ then $Q$ is free-connex acyclic.*

To also show completeness, we will interpret a GJT $T$ for a GCQ $Q$ as a "parse tree" that specifies the two-stage sequence of reduction steps that can be done on $\mathcal{H}(Q)$ to reach the empty triplet. Not all GJTs will allows us to do so easily, however, and we will therefore restrict our attention to those GJTs that are *canonical*.

**Definition 14 (Canonical)** A GJT $T$ is *canonical* if:

1. its root is labeled by $\emptyset$;
2. every leaf node $n$ is the child of an internal node $m$ with $\mathit{var}(n) = \mathit{var}(m)$;
3. for all internal nodes $n$ and $m$ with $n \neq m$ we have $\mathit{var}(n) \neq \mathit{var}(m)$; and
4. for every edge $m \to n$ and all $\theta \in \mathit{pred}(m \to n)$ we have $\mathit{var}(\theta) \cap (\mathit{var}(n) \setminus \mathit{var}(m)) \neq \emptyset$.

A connex subset $N$ of $T$ is *canonical* if every node in it is an interior node of $T$. A GJT pair $(T, N)$ is canonical if both $T$ and $N$ are canonical.

The following proposition, proven in Appendix B, shows that we may restrict our attention to canonical GJT pairs without loss of generality.

**Proposition 9** *For every GJT pair there exists an equivalent canonical pair.*

We also require the following auxiliary notions and insights. First, if $(T, N)$ is a GJT pair, then define the *hypergraph associated to* $(T, N)$, denoted $\mathit{hyp}(T, N)$, to be the hypergraph formed by node labels in $N$,

$$\mathit{hyp}(T, N) = \{\mathit{var}_T(n) \mid n \in N, \mathit{var}_T(n) \neq \emptyset\}.$$

Further, define $\mathit{pred}(T, N)$ to be the set of all predicates occurring on edges between nodes in $N$. For a hyperedge $\overline{z}$, define the *hypergraph triplet* of $(T, N)$ w.r.t. $\overline{z}$, denoted $\mathcal{H}(T, N, \overline{z})$ to be the hypergraph triplet $(\mathit{hyp}(T, N), \overline{z}, \mathit{pred}(T, N))$.

The following technical Lemma shows that we can use canonical pairs as "parse" trees to derive a sequence of reduction steps. Its proof can be found in Appendix B.

**Lemma 5** *Let $(T, N_1)$ and $(T, N_2)$ be canonical GJT pairs with $N_2 \subseteq N_1$. Then $\mathcal{H}(T, N_1, \overline{z}) \rightsquigarrow^* \mathcal{H}(T, N_2, \overline{z})$ for every $\overline{z} \subseteq \mathit{var}(N_2)$.*

We require the following additional lemma, proven in Appendix B:

**Lemma 6** *Let $H_1$ and $H_2$ be two hypergraphs such that for all $e \in H_2$ there exists $\ell \in H_1$ such that $e \subseteq \ell$. Then $(H_1 \cup H_2, \overline{z}, \Theta) \rightsquigarrow^* (H_1, \overline{z}, \Theta)$, for every hyperedge $\overline{z}$ and set of predicates $\Theta$.*

We these tools in hand we can prove completeness.

**Proposition 10** *Let $Q$ be a GCQ, let $T$ be a GJT for $Q$ and let $N$ be a connex subset of $T$ with $\mathit{out}(Q) \subseteq \mathit{var}(N)$. Assume that $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}\!\downarrow$ and $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}\!\downarrow$. Then $\mathcal{J}$ is the empty triplet and $\mathit{var}(\mathit{hyp}(\mathcal{I})) \subseteq \mathit{var}(N)$.*

*Proof.* By Proposition 9 we may assume without loss of generality that $(T, N)$ is a canonical GJT pair. Let $A$ be the set of all of $T$'s interior nodes. Clearly, $A$ is a connex subset of $T$ and $var(A) \subseteq var(Q)$. Furthermore, because for every atom $r(\overline{x})$ in $Q$ there is a leaf node $l$ in $T$ labeled by $r(\overline{x})$ (as $T$ is a GJT for $Q$), which has a parent interior node $n_l$ labeled $\overline{x}$ (because $T$ is canonical), also $var(Q) \subseteq var(A)$. Therefore, $var(A) = var(Q)$. By the same reasoning, $hyp(Q) \subseteq hyp(T, A)$. Therefore, $hyp(T, A) = hyp(T, A) \cup hyp(Q)$. Furthermore, because every interior node in a GJT has a guard descendant, and the leaves of $T$ are all labeled by atoms in $Q$, we know that for every node $n \in A$ there exists some hyperedge $f \in hyp(Q)$ such that $var(n) \subseteq var(f)$. In addition, we claim that $pred(T, A) = pred(Q)$. Indeed, $pred(T, A) \subseteq pred(Q)$ since $T$ is a GJT for $Q$. The converse inclusion follows from canonicality properties (2) and (4): because leaf nodes in a canonical GJT have a parent labeled by the same hyperedge, there can be no predicates on edges to leaf nodes in $T$. Thus, all predicates in $T$ are on edges between interior nodes, i.e., in $pred(T, A)$. Then, because every predicate in $Q$ appears somewhere in $T$ (since $T$ is a GJT for $Q$), we have $pred(Q) \subseteq pred(T, A)$. From all of the observations made so far and Lemma 6, we obtain:

$$\begin{aligned}
&\mathcal{H}(T, A, out(Q)) \\
&= (hyp(T, A), out(Q), pred(T, A)) \\
&= (hyp(T, A) \cup hyp(Q), out(Q), pred(T, A)) \\
&\overset{*}{\rightsquigarrow} (hyp(Q), out(Q), pred(T, A)) \\
&= (hyp(Q), out(Q), pred(Q)) = \mathcal{H}(Q)
\end{aligned}$$

Thus $\mathcal{H}(T, A, out(Q)) \rightsquigarrow^* \mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}$. Furthermore, because $(T, N)$ is also canonical with $N \subseteq A$ and $out(Q) \subseteq var(N)$ we have $\mathcal{H}(T, A, out(Q)) \rightsquigarrow^* \mathcal{H}(T, N, out(Q))$ by Lemma 5. Then, because reduction is confluent (Proposition 7) we obtain that $\mathcal{H}(T, N, out(Q))$ and $\mathcal{I}$ can be reduced to the same triplet. Because $\mathcal{I}$ is in normal form, necessarily $\mathcal{H}(T, N, out(Q)) \rightsquigarrow^* \mathcal{I}$. Since reduction steps can only remove nodes and hyperedges (and never add them), $var(hyp(\mathcal{I})) \subseteq var(N)$.

It remains to show that $\mathcal{J}$ is the empty triplet. Hereto, first verify the following. For any hypergraph triplets $\mathcal{U}$ and $\mathcal{V}$, if $\mathcal{U} \rightsquigarrow^* \mathcal{V}$ then also $\tilde{\mathcal{U}} \rightsquigarrow^* \tilde{\mathcal{V}}$. From this, $\mathcal{H}(T, A, out(Q)) \rightsquigarrow^* \mathcal{I}$, and the fact that $\mathcal{H}(T, A, \emptyset)$ is the residual of $\mathcal{H}(T, A, out(Q))$ we conclude $\mathcal{H}(T, A, \emptyset) \rightsquigarrow^* \tilde{\mathcal{I}}$. Then, because $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}$, it follows that $\mathcal{H}(T, A, \emptyset) \rightsquigarrow^* \mathcal{J}$. Let $r$ be $T$'s root node, which is labeled by $\emptyset$ since $T$ in canonical. Then $\{r\}$ is a connex subset of $T$. By Lemma 5, $\mathcal{H}(T, A, \emptyset) \rightsquigarrow^* \mathcal{H}(T, \{r\}, \emptyset)$. Now observe that the hypergraph of $\mathcal{H}(T, \{r\}, \emptyset)$ is empty, and its predicate set is also empty. Therefore, $\mathcal{H}(T, \{r\}, \emptyset)$ is the empty hypergraph triplet. In particular, it is in normal form. But, since $\mathcal{J}$ is also in normal form and normal forms are unique, $\mathcal{J}$ must also be the empty triplet. □

**Corollary 2 (Completeness)** *Let $Q$ be a GCQ. Assume that $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}{\downarrow}$ and $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}{\downarrow}$.*

1. *If $Q$ is acyclic, then $\mathcal{J}$ is the empty triplet.*
2. *If $Q$ is free-connex acyclic, then $\mathcal{J}$ is the empty triplet and $var(hyp(\mathcal{I})) = out(Q)$.*
3. *For every GJT $T$ of $Q$ and every connex subset $N$ of $T$ it holds that $var(hyp(\mathcal{I})) \subseteq var(N)$.*

*Proof.* (1) Since $Q$ is acyclic, there exists a GJT $T$ for $Q$. Let $N$ be the set of all of $T$'s nodes. Then $N$ is a connex subset of $T$ and $out(Q) \subseteq var(N) = var(Q)$. The result then follows from Proposition 10.

(2) Since $Q$ is free-connex acyclic, there exists a GJT pair $(T, N)$ compatible with $Q$. In particular, $var(N) = out(Q)$. By Proposition 10, $\mathcal{J}$ is the empty triplet, and $var(hyp(\mathcal{I})) \subseteq var(N) = out(Q)$. It remains to show $out(Q) \subseteq var(hyp(\mathcal{I}))$. First verify the following: A reduction step on a hypergraph triplet $\mathcal{H}$ never removes any variable in $out(\mathcal{H})$ from $hyp(\mathcal{H})$, nor does it modify $out(\mathcal{H})$. Then, since $out(\mathcal{H}(Q)) = out(Q) \subseteq var(Q) \subseteq var(hyp(\mathcal{H}(Q))))$, and $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}$ we obtain $out(Q) \subseteq var(hyp(\mathcal{I}))$.

(3) Follows directly from Proposition 10. □

Theorem 4 follows directly from Corollaries 1 and 2. Theorem 5 follows from Theorem 4 and Proposition 8.

### 5.2 Transforming GJT pairs to query plans

Let us call a GJT pair $(T, N)$ *binary* if $T$ is binary, and *sibling-closed* if $N$ is sibling-closed. A query plan is hence a binary and sibling-cloded GJT pair. In this section, we prove the following result.

**Proposition 11** *Every GJT pair can be transformed in polynomial time into an equivalent plan.*

We prove Proposition 11 in two steps. First, we show that any pair $(T, N)$ can be transformed in polynomial time into an equivalent sibling-closed pair. Next, we show that any sibling-closed GJT pair $(T, N)$ can be converted in polynomial time into an equivalent plan. Proposition 11 hence follows by composing these two transformations. Throughout this section, let $\mathrm{ch}_T(n)$ denote the set of children of $n$ in $T$.

**Sibling-closed transformation.** We say that $n \in T$ is a *violator* node in a GJT pair $(T, N)$ if $n \in N$ and some, but not all children of $n$ are in $N$. A violator is of *type 1* if some node in $\mathrm{ch}_T(n) \cap N$ is a guard of $n$. It is of *type 2* otherwise. We now define two operations

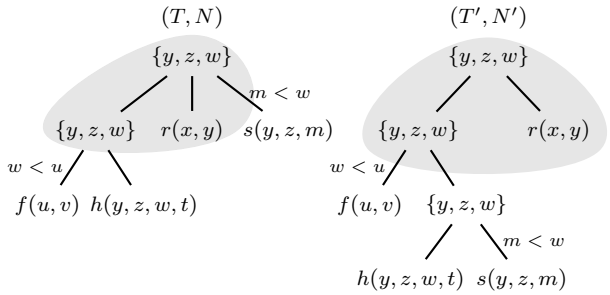


**Fig. 7** Illustration of the sibling-closed transform: removal of type-1 violator. The connex sets $N$ and $N'$ are indicated by the shaded areas.

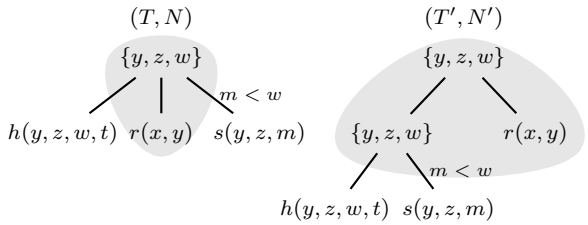


**Fig. 8** Illustration of the sibling-closed transform: removal of type-2 violator. The connex sets $N$ and $N'$ are indicated by the shaded areas.

on $(T, N)$ that remove violators of type 1 and type 2, respectively. The sibling-closed transformation is then obtained by repeatedly applying these operators until all violators are removed.

The first operator is applicable when $n$ is a type 1 violator. It returns the pair $(T', N')$ obtained as follows:

- Since $n$ is a type 1 violator, some $g \in \mathrm{ch}_T(n) \cap N$ is a child guard of $n$ (i.e., $var(n) \subseteq var(g)$).
- Because every node has a guard, there is some leaf node $l$ that is a descendant guard of $g$ (i.e. $var(g) \subseteq var(l)$). Possibly, $l$ is $g$ itself.
- Now create a new node $p$ between node $l$ and its parent with label $var(p) = var(l)$. Since $l$ is a descendant guard of $n$ and $g$, $p$ becomes a descendant guard of $n$ and $g$ as well. Detach all nodes in $\mathrm{ch}_T(n) \setminus N$ from $n$ and attach them as children to $p$, preserving their edge labels. This effectively moves all subtrees rooted at nodes in $\mathrm{ch}_T(n) \setminus N$ from $n$ to $p$. Denote by $T'$ the final result.
- If $l$ was not in $N$, then $N' = N$. Otherwise, $N' = N \setminus \{l\} \cup \{p\}$.

We write $(T, N) \xrightarrow{1,n} (T', N')$ to indicate that $(T', N')$ can be obtained by applying the above-described operation on node $n$.

*Example 7* Consider the GJT pair $(T, N)$ from Fig. 7 where $N$ is indicated by the nodes in the shaded area. Let us denote the root node by $n$ and its guard child with label $\{y, z, w\}$ by $g$. The node $l = h(y, z, w, t)$ is a descendant guard of $g$. Since $s(y, z, m)$ is not in $N$, $n$ is violator of type 1. After applying the operation 1 for the choice of guard node $g$ and descendant guard node $l$, $(T', N')$ shows the resulting valid sibling-closed GJT.

**Lemma 7** *Let $n$ be a violator of type 1 in $(T, N)$ and assume $(T, N) \xrightarrow{1,n} (T', N')$. Then $(T', N')$ is a GJT pair and it is equivalent to $(T, N)$. Moreover, the number of violators in $(T', N')$ is strictly smaller than the number of violators in $(T, N)$.*

We prove this lemma in Appendix C. The second operator is applicable when $n$ is a type 2 violator. When applied to $n$ in $(T, N)$ it returns the pair $(T', N')$ obtained as follows:

- Since $n$ is a type 2 violator, no node in $\mathrm{ch}_T(n) \cap N$ is a guard of $n$. Since every node has a guard, there is some $g \in \mathrm{ch}_T(n) \setminus N$ which is a guard of $n$.
- Create a new child $p$ of $n$ with label $var(p) = var(n)$; detach all nodes in $\mathrm{ch}_T(n) \setminus N$ (including $g$) from $N$, and add them as children of $p$, preserving their edge labels. This moves all subtrees rooted at nodes in $\mathrm{ch}_T(n) \setminus N$ from $n$ to $p$. Denote by $T'$ the final result.
- Set $N' = N \cup \{p\}$.

We write $(T, N) \xrightarrow{2,n} (T', N')$ to indicate that $(T', N')$ was obtained by applying this operation on $n$.

*Example 8* Consider the GJT pair $(T, N)$ in Fig. 8. Let us denote the root node by $n$. Since its guard child $h(y, z, w, t)$ is not in $N$, $n$ is violator of type 2. After applying operation 2 on $n$, $(T', N')$ shows the resulting valid sibling-closed GJT.

**Lemma 8** *Let $n$ be a violator of type 2 in $(T, N)$ and assume $(T, N) \xrightarrow{2,n} (T', N')$. Then $(T', N')$ is a GJT pair and it is equivalent to $(T, N)$. Moreover, the number of violators in $(T', N')$ is strictly smaller than the number of violators in $(T, N)$.*

The proof can be found in Appendix C.

**Proposition 12** *Every GJT pair can be transformed in polynomial time into an equivalent sibling-closed pair.*

*Proof.* The two operations introduced above remove violators, one at a time. By repeatedly applying these operations until no violator remains we obtain an equivalent pair without violators, which must hence be sibling-closed. Since each operator can clearly be executed in polynomial time and the number of times that we must apply an operator is bounded by the number of nodes in the GJT pair, the removal takes polynomial time. □

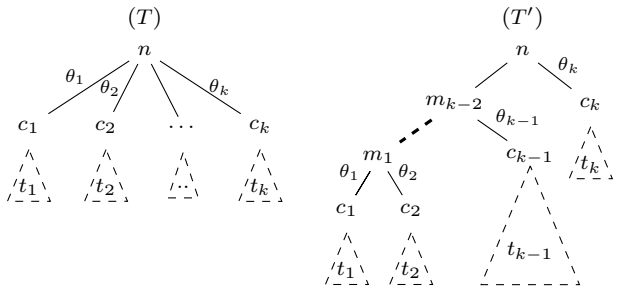


**Fig. 9** Binarizing a $k$-ary node $n$.

**Binary transformation.** Next, we show how to transform a sibling-closed pair $(T, N)$ into an equivalent binary and sibling-closed pair $(T', N')$. The idea here is to "binarize" each node $n$ with $k > 2$ children as shown in Fig. 9. There, we assume without loss of generality that $c_1$ is a guard child of $n$. The binarization introduces $k-2$ new intermediate nodes $m_1, \ldots, m_{k-2}$, all with $var(m_i) = var(n)$. Note that, since $c_1$ is a guard of $n$ and $var(m_i) = var(n)$, it is straightforward to see that $c_1$ will be a guard of $m_1$, which will be a guard of $m_2$, which will be a guard of $m_3$, and so on. Finally, $m_{k-2}$ will be a guard of $n$. The connex set $N$ is updated as follows. If none of $n$'s children are in $N$ i.e. $n$ is a frontier node, set $N' = N$. Otherwise, since $N$ is sibling-closed, all children of $n$ are in $N$, and we set $N' = N \cup \{m_1, \ldots, m_{k-2}\}$. Clearly, $N'$ remains a sibling-closed connex subset of $T'$ and $var(N') = var(N)$. We may hence conclude:

**Lemma 9** *By binarizing a single node in a sibling-closed GJT pair $(T, N)$ as shown in Fig. 9, we obtain an equivalent GJT pair $(T', N')$ that has strictly fewer non-binary nodes than $(T, N)$.*

Binarizing a single node is a polynomial-time operation. Then, by iteratively binarizing non-binary nodes until all nodes have become binary we hence obtain:

**Proposition 13** *Every sibling-closed GJT pair can be transformed in polynomial time into an equivalent plan.*

## 6 Implementation

We have implemented IEDyn, the instantiation of GDyn to setting where all $\theta$-joins are inequality joins described in Section 4.4, as a query compiler that generates executable code in the Scala programming language. The generated code instantiates a $(T, N)$-rep and defines *trigger functions* that maintain the $(T, N)$-rep under updates. Off-the-shelf Scala collection libraries are used to implement the required indexes, and we take care to share the data structures between the join and enumeration indexes whenever possible. An important optimization used by our implementation lies in observing that, for the nodes in the connex set $N$ that are not in the frontier of $N$ we never use the multiplicities stored in $\rho_n$ during enumeration. As such, we also do not need to compute these multiplicities during update processing. While our theoretical framework supports batch updates, our implementation is currently limited to single-tuple updates.

Our implementation supports two modes of operation: *push-based* and *pull-based*. In both modes, the system maintains the $T$-rep under updates. In the *push-based mode* the system generates, on its output stream, the delta result $\Delta Q(db, u)$ after each single-tuple update $u$. To do so, it uses a modified version of enumeration (Algorithm 1) that we call *delta enumeration*. Similarly to how Algorithm 1 enumerates $Q(db)$, delta enumeration enumerates $\Delta Q(db, u)$ with constant delay (if $Q$ has at most one inequality per pair of atoms) resp. logarithmic delay (otherwise). To do so, it uses both (1) the $T$-reduct GMRs $\rho_n$ and (2) the delta GMRs $\Delta\rho_n$ that are computed by Algorithm 1 when processing $u$. In this case, however, one also needs to index the $\Delta\rho_n$ similarly to $\rho_n$. In the *pull-based mode*, in contrast, the system only maintains the $(T, N)$-rep under updates but does not generate any output stream. Nevertheless, at any time a user can ENUM (Algorithm 1) to obtain the current output.

We have described in Section 4 how IEDyn can process free-connex acyclic GCQs under updates. It should be noted that our implementation also supports the processing of general acyclic GCQs that are not necessarily free-connex. This is done using the following simple strategy. Let $Q$ be acyclic but not free-connex. First, compute a free-connex acyclic approximation $Q_F$ of $Q$. $Q_F$ can always be obtained from $Q$ by extending the set of output variables of $Q$. In the worst case, we need to add all variables, and $Q_F$ becomes the full join underlying $Q$. Then, use IEDyn to maintain a $(T, N)$-rep for $Q_F$. When operating in push-based mode, for each update $u$, we use the $(T, N)$-rep to delta-enumerate $\Delta Q_F(db, u)$ and project each resulting tuple to materialize $\Delta Q(db, u)$ in an array. Subsequently, we copy this array to the output. Note that the materialization of $\Delta Q(db, u)$ here is necessary since the delta enumeration can produce duplicate tuples after projection. When operating in pull-based mode, we materialize $Q(db)$ in an array, and use delta enumeration of $Q_F$ to maintain the array under updates. Of course, under this strategy, we require $\Omega(|Q(db)|)$ space in the worst case, just like (H)IVM would, but we avoid the (partial) materialization of delta queries. Note the distinction between the two modes: in push-based mode $\Delta Q(db, u)$ is materialized (and discarded once the output is generated), while in pull-based mode $Q(db)$ is materialized upon requests.

| # | Query expression | Features | | | |
|---|---|---|---|---|---|
| | | $<$ | $=$ | $\pi$ | FC |
| $Q_1$ | $R(a,b,c), S(d,e,f) \mid a < d$ | ✓ | | | ✓ |
| $Q_2$ | $R(a,b,c,k), S(d,e,f,k) \mid a < d$ | ✓ | ✓ | | ✓ |
| $Q_3$ | $R(a,b,c), S(d,e,f), T(g,h,i) \mid a < d \wedge e < g$ | ✓ | | | ✓ |
| $Q_4$ | $R(a,b,c), S(d,e,f), T(g,h,i) \mid a < d \wedge d < g$ | ✓ | | | ✓ |
| $Q_5$ | $R(a,b,c,k), S(d,e,f,k), T(g,h,i) \mid a < d \wedge d < g$ | ✓ | ✓ | | ✓ |
| $Q_6$ | $R(a,b,c), S(d,e,f,k), T(g,h,i,k) \mid a < d \wedge d < g$ | ✓ | ✓ | | ✓ |
| $Q_7$ | $R(a,b,c,k), S(d,e,f,k), T(g,h,i,k) \mid a < d \wedge d < g$ | ✓ | ✓ | | ✓ |
| $Q_8$ | $\pi_{a,b,d,e,f,g,h}(Q_4)$ | ✓ | | ✓ | ✓ |
| $Q_9$ | $\pi_{a,d,e,f,g,h,k}(Q_5)$ | ✓ | ✓ | ✓ | ✓ |
| $Q_{10}$ | $\pi_{d,e,f,g,h,k}(Q_6)$ | ✓ | ✓ | ✓ | ✓ |
| $Q_{11}$ | $\pi_{a,b,d,e,g,h,k}(Q_7)$ | ✓ | ✓ | ✓ | ✓ |
| $Q_{12}$ | $\pi_{b,c,e,f,h,i}(Q_4)$ | ✓ | | ✓ | |
| $Q_{13}$ | $\pi_{b,c,e,f,h,i}(Q_5)$ | ✓ | ✓ | ✓ | |
| $Q_{14}$ | $\pi_{b,c,e,f,h,i}(Q_6)$ | ✓ | ✓ | ✓ | |
| $Q_{15}$ | $\pi_{b,c,e,f,h,i}(Q_7)$ | ✓ | ✓ | ✓ | |

**Table 1** Benchmark queries. FC = Free-connex acyclic.

Our query compiler computes query plans using the algorithm of Section 5. Whenever we have the choice between enacting multiple reduction steps, we first enact using (ISO), then using (FLT), and finally using (CSE). This corresponds to the usual heuristics of pushing down projections and selections. If multiple applications of (CSE) are possible, we prefer those where the hyperedge to be eliminated has no extended variables. This corresponds to pushing down equi-semijoins so that inequality semijoins are hopefully executed over GMRs of reduced cardinality.

## 7 Experimental Evaluation

In this section, we experimentally compare IEDyn against competing state-of-the art HIVM and CER systems.

**Queries.** Because the effectiveness of Dyn for equijoin queries has already been documented [25], we focus on inequality-join queries during our experimental analysis. Since there is no industry-strength established benchmark suite for such queries, we perform a systematic and in-depth exploration of the design space of inequality-joins of up to three relations. Concretely, we evaluate IEDyn on the queries listed in Table 1. Here, $Q_1$–$Q_7$ are full join queries (i.e., queries without projections). Among these, $Q_1$, $Q_3$ and $Q_4$ contain only inequality-join predicates, while $Q_2$, $Q_5$–$Q_7$ additionally contain at least one equality-join. Queries $Q_1$ and $Q_2$ are binary joins, while $Q_3$–$Q_6$ are multi-way join queries over three relations. The inequality predicates in $Q_3$ are unrelated ($a < d \wedge e < g$) while they form a chain ($a < d < g$) in $Q_4$–$Q_7$. Note that $Q_5$–$Q_7$ are variants of $Q_4$ with equijoins added. We have similarly experimented with variants of $Q_3$ with equijoins added, but the trends are similar to what we obtain for $Q_4$–$Q_7$. Collectively, $Q_3$–$Q_7$ (and the omitted variants) cover all possible ways in which three relations can be inequality-joined in an acyclic manner.

Queries $Q_8$–$Q_{15}$ project over the result of queries $Q_4$–$Q_7$. Among these, $Q_8$–$Q_{11}$ are free-connex acyclic while $Q_{12}$–$Q_{15}$ are acyclic but not free-connex.

**Updates.** We evaluate on streams of synthetic updates where each update consists of a single tuple insertion. We focus on the setting where updates are single tuple insertions for the following reasons. First, single tuple updates stress-test dynamic query processing since the query results must be kept up-to-date after each and every single tuple, in contrast to the setting for batch updates, where results can be out-of-sync for the duration of the batch. Second, since batch updates can always be processed by executing all updates in the batch individually (using the single-tuple update triggers), performance measurements for single-tuple updates yield an upper bound for the performance of batch updates. Third, while it is true that, in principle, the processing of batch updates can be done faster than that of single tuple updates (e.g., by amortizing index access and data structure updates over the entire batch instead of per tuple), both our implementation and most of the competing systems that we describe below do not implement such optimization.[6] As such, we restrict attention to single-tuple updates. We focus on insertions because this is supported by all of the systems that we consider whereas explicit deletions are not. While we have experimented with mixed (insert and delete) update streams for IEDyn, performance is similar to that for insert-only streams. This is expected, since IEDyn treats insertions and deletions uniformly.

The database is always empty when we start processing the update stream. We synthetically generate two kinds of update streams: *randomly-ordered* and *temporally-ordered* update streams. In *randomly-ordered* update streams, insertions can occur in any order. In contrast, *temporally-ordered* update streams guarantee that any attribute that participates in an inequality in the query has a larger value than the same attribute in any of the previously inserted tuples. Randomly-ordered update streams are useful for comparing against systems that allow processing of out-of-order tuples; temporally-ordered update streams are useful for comparison against systems that assume events arrive always with increasing timestamp values. Examples of systems that process temporally-ordered streams are automaton-based CER systems.

**Competitors.** We compare IEDyn with the following state-of-the art HIVM and CER engines: DBToaster

[6] In the sense that batch updates are only supported by treating each update tuple in the batch individually.

| | Mode | Stream | Unsupported queries |
|---|---|---|---|
| DBT | Pull | Random | |
| E | Push | Random | |
| SE | Push | Ordered | $Q_3$, $Q_5$–$Q_{15}$ |
| T | Push | Ordered | $Q_3$ |
| Z | Push | Ordered | $Q_8$–$Q_{15}$ |

**Table 2** Competing systems capabilities overview.

(DBT) [30], Esper (E) [19], SASE (SE) [3, 46, 49], Tesla (T) [14, 15], and ZStream (Z) [31]. The competing systems differ in their mode of operation (push-based vs pull-based, cf. Section 6) and some of them only support temporally-ordered streams. The capabilities of each system is summarized in Table 2, and discussed in detail in Appendix D.

**Setup.** Our experiments are run on an 8-core 3.07 GHz machine running Ubuntu with GNU/Linux 3.13.0-57-generic. To compile the different systems or generated trigger programs, we have used GCC version 4.8.2, Java 1.8.0_101, and Scala version 2.12.4. Each query is evaluated 10 times to measure update processing delay, and two times to measure memory footprint. We present the average over those runs. Each time a query is evaluated, 20 GB of main memory are freshly allocated to the program. To measure the memory footprint for Scala/Java based systems, we invoke the JVM system calls every 10 updates and consider the maximum value. For C/C++ based systems we use the GNU/Linux *time* command to measure memory usage. Experiments that measure memory footprint are always run separately of the experiments that measure processing time.

### 7.1 Results

We will compare using *total update processing time*, *memory footprint*, and *enumeration delay* as metrics. Here, the total update processing time is the time required to process the entire update stream, where updates are fed into the systems as fast as they can process them. This hence measures the maximum system performance. While measuring the update processing time, we take care to compare fairly with the competing systems by consistently running IEDyn in the same operation mode as the one supported by the competitor. Concretely, for push-based systems we report the time required to process the entire update stream, while generating the delta result $\Delta Q(db, u)$ after each update (cf. Section 6). When comparing against a pull-based system, the measured time includes only processing the entire update stream. For these systems, we later separately report the enumeration delay, i.e., the speed with which the result can be generated from the underlying representation of the output (a $T$-representation in the case of IEDyn). SASE, ZStream, and Tesla only support temporally-ordered streams and hence we only compare with them on such streams. While DBToaster and Esper support both random and temporally-ordered streams, we only report comparisons using randomly-ordered streams. We have also looked at temporally-ordered streams for these systems, but their throughput is similar (fluctuating between 3% and 12%) while that of IEDyn significantly improves (fluctuating between 35% and 50%) because insertions to sorted lists become constant instead of logarithmic. We omit these experiments due to lack of space.

Some executions of the competing systems failed either because they required more than 20GB of main memory or they took more than 1500 seconds. If an execution requires more than 20GB, we report the processing time elapsed until the exception was raised. If an execution is still running after 1500 seconds, we stop it and report its maximum memory usage while running.

**Full join queries.** We first analyze the performance of the full join queries $Q_1$–$Q_7$ in the setting where the join selectivity is relatively large. Here, the selectivity of $R \bowtie S$ is defined as $\frac{|R \bowtie S|}{|R| \times |S|}$. High-selectivity joins are common in CER scenarios. We refer to Table 3(Left) for output sizes. Fig. 10 compares the update processing time of IEDyn against the competing systems for full join queries $Q_1$–$Q_7$, grouped per system capability (push/pull, random/temporal). We observe that all of the competing systems have large processing times even for very small update stream sizes, indicating poor scalability. In particular, $DBT$ runs out of memory on streams of sizes $\geq 18k$ for query $Q_5$, and on streams of size $\geq 15k$ for query $Q_6$. $Z$ runs out of memory for $Q_5$ and $Q_6$ on streams of size $21k$. Similarly, $T$ took more than 1500 seconds for $Q_5$ on streams of size $\geq 12k$, for $Q_6$ on streams of size $\geq 18k$, and for $Q_7$ on streams of size $\geq 9k$ and was aborted at that time. All of these behaviors are due to the large selectivity of joins on this dataset. Note that in all cases, IEDyn scales satisfactorily with increasing stream sizes, and significantly outperforms the competitors, often by orders of magnitude. This is confirmed in Table. 3(Right) where we show the processing time and memory footprint used by IEDyn as a percentage of the corresponding usage in the competing systems (for the largest stream size of each query). Writing "$(x, y)$ oom": to indicate $x$ orders of magnitude improvement in processing time and $y$ orders in memory consumption, we see that IEDyn improves up to $(1, 2)$ oom w.r.t. Z; almost $(3, 1)$ oom w.r.t. T; almost $(2, 2)$ oom w.r.t SE, and up to $(4, 3)$ oom w.r.t DBT. Moreover, for these queries, even in

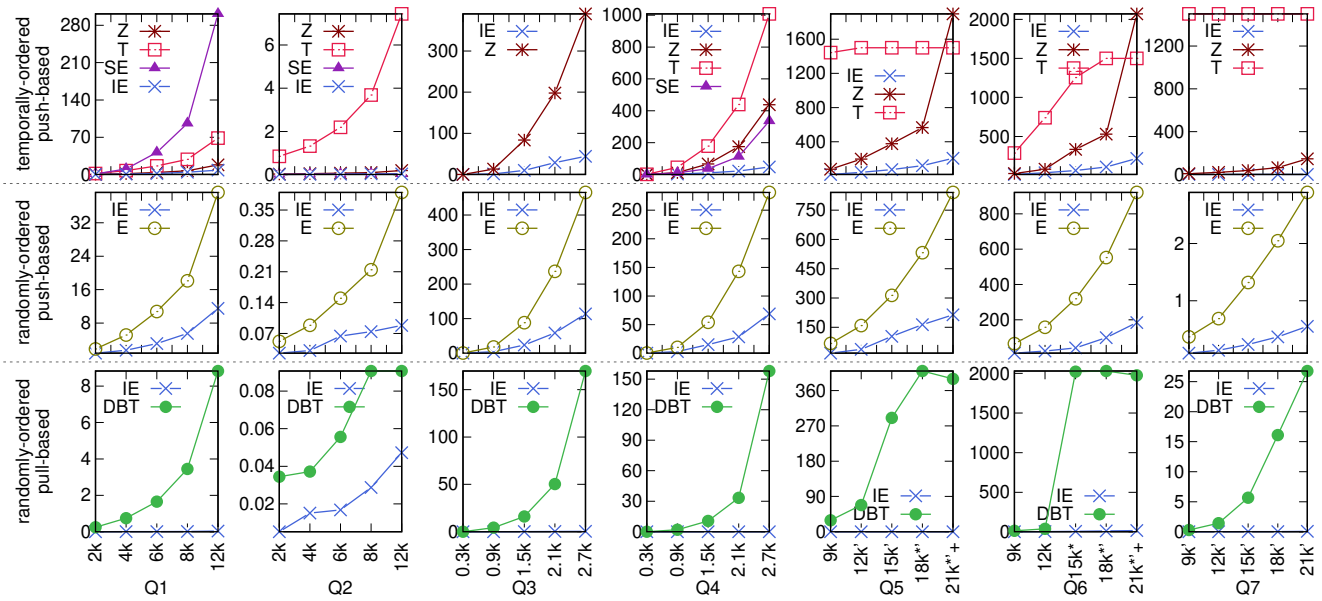


**Fig. 10** Processing time in seconds on full join queries for various stream sizes. (*: *DBT* out of memory, +: *Z* out of memory, ′: *T* was stopped after 1500 seconds).

| | Data size | |
|---|---|---|
| | Stream | Output |
| $Q_1$ | $12k$ | $18,017k$ |
| $Q_2$ | $12k$ | $3.8k$ |
| $Q_3$ | $2.7k$ | $178,847k$ |
| $Q_4$ | $2.7k$ | $90,425k$ |
| $Q_5$ | $21k$ | $411,669k$ |
| $Q_6$ | $21k$ | $297,873k$ |
| $Q_7$ | $21k$ | $64,603k$ |
| $Q_8$ | $2.7k$ | $114,561k$ |
| $Q_9$ | $21k$ | $411,669k$ |
| $Q_{10}$ | $21k$ | $99,043k$ |
| $Q_{11}$ | $21k$ | $43,564k$ |
| $Q_{12}$ | $2.7k$ | $114,561k$ |
| $Q_{13}$ | $21k$ | $294,139k$ |
| $Q_{14}$ | $21k$ | $297,873k$ |
| $Q_{15}$ | $21k$ | $50,468k$ |

| | Z | | T | | SE | | E | | DBT | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Time | Mem | Time | Mem | Time | Mem | Time | Mem | Time | Mem |
| $Q_1$ | 45.33 | 3.40 | 12.06 | 96.15 | 2.74 | 25.23 | 29.33 | 24.51 | 0.43 | 2.74 |
| $Q_2$ | 19.34 | 87.10 | 0.52 | 168.75 | 97.80 | 1.00 | 22.62 | 16.67 | 52.10 | 0.20 |
| $Q_3$ | 11.30 | 0.87 | N/A | N/A | N/A | N/A | 24.49 | 16.57 | 0.25 | 0.19 |
| $Q_4$ | 10.69 | 0.69 | 4.64 | 3.45 | 13.94 | 55.56 | 24.50 | 14.20 | 0.01 | 0.20 |
| $Q_5$ | 10.85 | 0.30 | 13.72 | 19.11 | N/A | N/A | 25.50 | 23.44 | 0.04 | 0.15 |
| $Q_6$ | 10.58 | 0.31 | 14.61 | 11.03 | N/A | N/A | 23.28 | 17.37 | 0.86 | 0.15 |
| $Q_7$ | 0.03 | 2.38 | 0.05 | 24.10 | N/A | N/A | 19.18 | 67.14 | 0.32 | 1.18 |
| $Q_8$ | N/A | N/A | 9.01 | 6.36 | N/A | N/A | 24.91 | 10.24 | 0.02 | 0.35 |
| $Q_9$ | N/A | N/A | 13.96 | 40.00 | N/A | N/A | 34.16 | 94.34 | 0.07 | 0.25 |
| $Q_{10}$ | N/A | N/A | 0.07 | 38.52 | N/A | N/A | 17.49 | 23.38 | 24.22 | 61.84 |
| $Q_{11}$ | N/A | N/A | 0.04 | 10.23 | N/A | N/A | 4.95 | 25.43 | 0.01 | 0.45 |
| $Q_{12}$ | N/A | N/A | 7.11 | 10.49 | N/A | N/A | 21.42 | 14.17 | 71.19 | 54.07 |
| $Q_{13}$ | N/A | N/A | 9.99 | 47.58 | N/A | N/A | 29.59 | 109.26 | 10.14 | 68.29 |
| $Q_{14}$ | N/A | N/A | 13.96 | 47.20 | N/A | N/A | 34.68 | 28.92 | 91.75 | 77.90 |
| $Q_{15}$ | N/A | N/A | 0.16 | 25.26 | N/A | N/A | 95.80 | 70.00 | 0.17 | 15.02 |

**Table 3** (Left) Maximum update stream and result sizes, $k = 10^3$. (Right) Relative performance of IEDyn, expressed as a percentage of the corresponding resources used by competing systems. N/A = Not Applicable.

push-based mode IEDyn can support the enumeration of query results from its data structures at any time while competing push-based systems have no such support. Hence, IEDyn is not only more efficient but also provides more functionality.

Drilling deeper into the specific queries, we see from Fig. 10 that while existing systems already perform poorly on the inequality-only binary join query $Q_2$, this is further aggravated when moving to the corresponding three-way join query $Q_3$ and $Q_4$: note that the plot for $Q_1$ shows streams sizes up to $12k$, whereas the plots for $Q_3$ and $Q_4$ only go to $2.7k$ while having much larger runtimes. Because $Q_2$, and $Q_5$–$Q_7$ modify $Q_1$, resp. $Q_4$ by adding equality join predicates, they have a smaller join selectivity. As a result, performance for these queries is much better, across all systems. (To see this, note that, for $Q_5$–$Q_6$ the first data point plotted has a significantly lower processing time while processing a data stream three times the size.) Similarly, we note that, while not visually apparent from Fig. 10 the performance of all systems on $Q_3$ is $10\% - 100\%$ slower compared to the same systems on $Q_4$. This is due to the fact that the two unrelated inequality predicates ($a < d \wedge e < g$) of $Q_3$ are linearly ordered in $Q_4$ ($a < d \wedge d < g$), which causes $Q_4$ to have higher selectivity, improving performance. From these observations we may conclude that the performance of all systems increases as the join selectivity decreases. Nevertheless, in contrast to

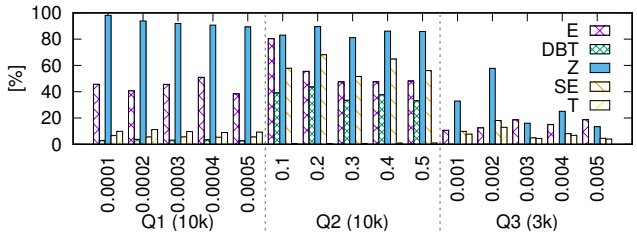


**Fig. 11** IEDyn as percentage of ($E$, $DBT$, $SE$, $Z$, $T$) for higher join selectivities. X-axis shows queries with tuples per relation and selectivities.

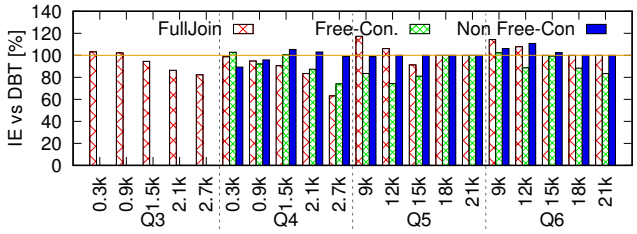


**Fig. 12** Enumeration of query results: IEDyn vs $DBT$ for various input stream sizes. Different bars for $Q_4, Q_5, Q_6$ show their projected versions.

existing systems, IEDyn continues to scale and perform satisfactorily even with large selectivities.

To confirm this trend in the setting of very low selectivities, were processing is hence less output-size dependent, we have also generated datasets with probability distributions that are parametrized by a *selectivity* $s$, such that the expected number of output tuples is $s$ percent of the cartesian product of all relations in the query. The results in Fig. 11 show that IEDyn consistently continues to perform better also on very selective inequality joins. For super selective inequality joins the measurements come similar to what we observe for equality joins, which we investigated in detail in [25].

**Projections.** The trends observed for full join queries are confirmed for the queries $Q_8$–$Q_{15}$ with projections, as shown in Table 3(Right). In particular, IEDyn improves up to almost $(4, 1)$ oom w.r.t. T; up to almost $(1, 1)$ w.r.t. $E$., and up to $(4, 3)$ oom w.r.t DBT. Again, we observe that, while IEDyn consistently outperforms the competitors, when the selectivity of the queries decreases, the performance gain of IEDyn increases. In addition, we note that the performance gain of IEDyn is bigger on queries that are free-connex acyclic (namely: $Q_8$–$Q_{10}$) as opposed to those that are not ($Q_{11}$–$Q_{14}$). This is to be expected since our implementation evaluates non-free-connex queries by approximating them by free-connex variants, requiring additional materialization (cf. Section 6).

**Result enumeration.** CDE is theoretical notion that hides a constant factor which could decrease performance in practice when compared to enumerating from a fully materialized representation In Fig. 12, we show the practical application of CDE in IEDyn and compare against $DBT$ which materializes the full query results. We plot the time required to enumerate the result from IEDyn's $T$-rep as a fraction of the time required to enumerate the result from $DBT$'s materialized views. As can be seen from the figure, both enumeration times are comparable on average. Note that we do not compare enumeration time for push-based systems, since for these systems the time required for delta enumeration is already included in the update processing time.

## 8 Conclusions

Traditional techniques for dynamic query evaluation are based on a trade-off between materialization of join subresults (to avoid recomputing these subresults) and their recomputation (to avoid the space overhead induced by materialization). We have shown that instead of materializing full join subresults, it suffices to maintain and index semijoin subresults. This methodology, called General Dynamic Yannakakis, allows us to maintain a data structure that, like recomputation, has low memory-overhead; and yet supports all operations one commonly expects from full materialization: enumeration with bounded delay and efficient processing of updates. In addition, the framework supports bounded-delay delta-enumeration under single-tuple update, hence allowing to operate in push-based mode similar to streaming systems. Our experiments against state-of-the art engines in the IVM and CER domains show that Dynamic Yannakakis can improve performance by orders of magnitude in both time and space. The performance gap with existing systems is the largest for output-dominated queries (i.e., queries where the join result is large). While the gap decreases for more selective joins, GDyn continues to consistently outperform existing systems. In addition, while GDyn's theory is developed only for free-connex acyclic GCQs, our experiments show that GDyn's adaptation to acyclic GCQs that are not-free connex (by means of free-connex approximation followed by post-processing) is equally effective compared to existing systems.

From a theoretical viewpoint, it would be satisfying to establish lower bounds on the complexity of processing inequality-joins dynamically. For equijoins, such lower bounds were recently established by Berkholz, Keppeler, and Schweikardt [8]. Since GDyn meets these lower bounds on queries with equijoins only [25], it would be interesting to know whether it is similarly optimal for inequality-joins.

## References


1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Addison-Wesley Longman Publishing Co., Inc., 1995.
2. M. Abo Khamis, H. Q. Ngo, and A. Rudra. FAQ: Questions asked frequently. In *Proc. of PODS*, pages 13–28, 2016.
3. J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proc. of SIGMOD 2008*, pages 147–160, 2008.
4. A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: the stanford data stream management system. In *Data Stream Management - Processing High-Speed Data Streams*, pages 317–336. 2016.
5. F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.
6. G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Proc. of CSL*, pages 208–222, 2007.
7. N. Bakibayev, T. Kočiský, D. Olteanu, and J. Závodný. Aggregation and ordering in factorised databases. *Proc. of VLDB*, 6(14):1990–2001, 2013.
8. C. Berkholz, J. Keppeler, and N. Schweikardt. Answering conjunctive queries under updates. In *Proc. of PODS*, pages 303–318, 2017.
9. P. A. Bernstein and N. Goodman. The power of inequality semijoins. *Inf. Syst.*, 6(4):255–265, 1981.
10. J. Brault-Baron. *De la pertinence de l'énumération: complexité en logiques*. PhD thesis, Université de Caen, 2013.
11. L. Brenna, A. J. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. M. White. Cayuga: a high-performance event processing engine. In *Proc. of SIGMOD 2007*, pages 1100–1102, 2007.
12. R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.
13. T. Cormen. *Introduction to Algorithms, 3rd Edition:*. MIT Press, 2009.
14. G. Cugola and A. Margara. TESLA: a formally defined event specification language. In *Proc. of DEBS 2010*, pages 50–61, 2010.
15. G. Cugola and A. Margara. Complex event processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012.
16. G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM CSUR*, 44(3):15:1–15:62, 2012.
17. D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *VLDB 1991*, pages 443–452, 1991.
18. J. Enderle, M. Hampel, and T. Seidl. Joining interval data in relational databases. In *Proc. of SIGMOD 2004*, pages 683–694, 2004.
19. EsperTech. Esper complex event processing engine. `http://www.espertech.com/`.
20. L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. VLDB*, pages 500–511, 2003.
21. A. Gupata and I. S. Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.
22. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. of SIGMOD*, pages 157–166, 1993.
23. J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB'95.*, pages 562–573, 1995.
24. M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proc. of STOC*, pages 21–30, 2015.
25. M. Idris, M. Ugarte, and S. Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proc. of SIGMOD 2017*, 2017.
26. M. Idris, M. Ugarte, S. Vansummeren, H. Voigt, and W. Lehner. Conjunctive queries with inequalities under updates. *PVLDB*, 11(7):733–745, 2018.
27. J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proc. of ICDE*, pages 341–352, 2003.
28. Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and P. Kalnis. Fast and scalable inequality joins. *VLDB J.*, 26(1):125–150, 2017.
29. C. Koch. Incremental query evaluation in a ring of databases. In *Proc. of PODS*, pages 87–98, 2010.
30. C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB Journal*, pages 253–278, 2014.
31. Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *Proc. of SIGMOD 2009*, pages 193–206, 2009.
32. M. Nikolic and D. Olteanu. Incremental view maintenance with triple lock factorization benefits. In *Proc. of SIGMOD 2018*, pages 365–380, 2018.
33. D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *ACM TODS*, 40(1):2:1–2:44, 2015.
34. P. Roy, J. Teubner, and R. Gemulla. Low-latency handshake join. *PVLDB*, 7(9):709–720, 2014.
35. B. Sahay and J. Ranjan. Real time business intelligence in supply chain analytics. *Information Management & Computer Security*, 16(1):28–48, 2008.
36. M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *Proc. of SIGMOD*, pages 3–18, 2016.
37. N. P. Schultz-Møller, M. Migliavacca, and P. R. Pietzuch. Distributed complex event processing with query rewriting. In *Proc. of DEBS 2009*, 2009.
38. L. Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Record*, 44(1):10–17, 2015.
39. M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, (4):42–47, 2005.
40. J. Teubner and R. Müller. How soccer players would do stream joins. In *Proc. of SIGMOD*, pages 625–636, 2011.
41. T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.
42. M. Y. Vardi. The complexity of relational query languages (extended abstract). In *Proc. of STOC*, pages 137–146, 1982.
43. S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proc. of VLDB*, pages 285–296, 2003.
44. W. Wang, J. Gao, M. Zhang, S. Wang, G. Chen, T. K. Ng, B. C. Ooi, J. Shao, and M. Reyad. Rafiki: Machine learning as an analytics service system. *PVLDB*, 12(2):128–140, 2018.

45. A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS 1991)*, pages 68–77. IEEE Computer Society, 1991.
46. E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proc. of SIGMOD 2006*, pages 407–418, 2006.
47. M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. of VLDB*, pages 82–94, 1981.
48. M. Yoshikawa and Y. Kambayashi. Processing inequality queries based on generalized semi-joins. In *VLDB*, pages 416–428, 1984.
49. H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in complex event processing. In *Proce. of SIGMOD*, 2014.

## A Proofs of Section 4

**Lemma 1** *$\rho_n = \mathcal{Q}[\![T_n, n]\!](db)$, for every node $n \in T$.*

*Proof.* We proceed by induction on the number of descendants of $n$. If $n$ has no descendants then $T_n$ is a single atom $r(\overline{x})$ with $\overline{x} = var(n) = out(\mathcal{Q}[\![T_n, n]\!])$. Then $\mathcal{Q}[\![T_n, n]\!](db) = (\pi_{var(n)} r(\overline{x}))(db) = r(\overline{x})(db) = db_{r(\overline{x})} = \rho_n$, concluding the basic case. Now, for the inductive case we distinguish whether $n$ has one or two children.

Assume $n$ has a single child $c$. Then $at(T_n) = at(T_c)$ and $pred(T_n) = pred(T_c) \cup pred(n)$. Therefore, by definition of $\mathcal{Q}[\![\cdot]\!]$, we have $\mathcal{Q}[\![T_n]\!] \equiv \sigma_{pred(n)} \mathcal{Q}[\![T_c]\!]$, which implies that $\mathcal{Q}[\![T_n, n]\!] = \pi_{var(n)} \mathcal{Q}[\![T_n]\!] \equiv \pi_{var(n)} \sigma_{pred(n)} \mathcal{Q}[\![T_c]\!]$. Furthermore, since $pred(n)$ only mentions variables in $var(c) \cup var(n)$ and $var(n) \subseteq var(c)$, as $c$ is a guard of $n$, this is equivalent to

$$\pi_{var(n)} \sigma_{pred(n)} \mathcal{Q}[\![T_c]\!] \equiv \pi_{var(n)} \sigma_{pred(n)} \pi_{var(c)} \mathcal{Q}[\![T_c]\!] = \pi_{var(n)} \sigma_{pred(n)} \mathcal{Q}[\![T_c, c]\!].$$

By induction, $\pi_{var(n)} \sigma_{pred(n)} \mathcal{Q}[\![T_c, c]\!](db) = \pi_{var(n)} \sigma_{pred(n)} \rho_c = \rho_n$, showing that $\mathcal{Q}[\![T_n, n]\!](db) = \rho_n$.

Assume now that $n$ has two children $c_1$ and $c_2$. We assume w.l.o.g. that $c_1$ is a guard for $n$. Note that $at(T_n) = at(T_{c_1}) \cup at(T_{c_2})$ and $pred(T_n) = pred(T_{c_1}) \cup pred(T_{c_2}) \cup pred(n)$. Therefore,

$$\mathcal{Q}[\![T_n]\!] \equiv \sigma_{pred(n)} \sigma_{pred(T_{c_1})} \sigma_{pred(T_{c_2})} \left(at(T_{c_1}) \bowtie at(T_{c_2})\right).$$

Here, we abuse notation and write $at(T_i)$ for the natural join of all atoms in $T_{c_i}$. Since $pred(T_{c_i})$ only mentions variables of atoms in $T_{c_i}$ (for $i \in \{1, 2\}$), we can push the selections:

$$\mathcal{Q}[\![T_n]\!] \equiv \sigma_{pred(n)} \left(\sigma_{pred(T_{c_1})} at(T_{c_1}) \bowtie \sigma_{pred(T_{c_2})} at(T_{c_2})\right) \equiv \sigma_{pred(n)} \left(\mathcal{Q}[\![T_{c_1}]\!] \bowtie \mathcal{Q}[\![T_{c_2}]\!]\right).$$

Therefore,

$$\mathcal{Q}[\![T_n, n]\!] = \pi_{var(n)} \mathcal{Q}[\![T_n]\!] \equiv \pi_{var(n)} \sigma_{pred(n)} \left(\mathcal{Q}[\![T_{c_1}]\!] \bowtie \mathcal{Q}[\![T_{c_2}]\!]\right).$$

Since $var(pred(n)) \subseteq var(c_1) \cup var(c_2) \cup var(n)$ and $var(n) \subseteq var(c_1)$ we have $var(pred(n)) \subseteq var(c_1) \cup var(c_2)$. This, combined with the fact that, due to the connectedness property of $T$ we, have $var(\mathcal{Q}[\![T_{c_1}]\!]) \cap var(\mathcal{Q}[\![T_{c_2}]\!]) \subseteq var(c_i)$ for $i \in \{1, 2\}$, we can add the following projections

$$\mathcal{Q}[\![T_n, n]\!] \equiv \pi_{var(n)} \sigma_{pred(n)} \left(\pi_{var(c_1)} \mathcal{Q}[\![T_{c_1}]\!] \bowtie \pi_{var(c_2)} \mathcal{Q}[\![T_{c_2}]\!]\right) \equiv \pi_{var(n)} \sigma_{pred(n)} \left(\mathcal{Q}[\![T_{c_1}, c_1]\!] \bowtie \mathcal{Q}[\![T_{c_2}, c_2]\!]\right).$$

Hence, by induction hypothesis we have

$$\mathcal{Q}[\![T_n, n]\!](db) = \pi_{var(n)} \sigma_{pred(n)} \left(\rho_{c_1} \bowtie \rho_{c_2}\right) = \rho_n,$$

concluding our proof. □

**Lemma 3** *1. $Q(db)$ is a positive GMR, for any GCQ $Q$ and any database $db$.*
*2. If $R$ is a positive GMR over $\overline{x}$ and $\overline{y} \subseteq \overline{x}$, then $\mathbf{t}[\overline{y}] \in \pi_{\overline{y}} R$ for every tuple $\mathbf{t} \in R$.*

*Proof.* (1) Follows by straightforward induction on $Q$, using the fact that the GMRs in $db$ are themselves positive by definition. (2) Is a standard result in relational algebra, which hence transfers to the case of positive GMRs. □

**Lemma 10** *Let $R$ be a positive GMR over $\overline{x}$, $S$ a positive GMR over $\overline{y}$ and $\mathbf{t}$ a tuple over $\overline{z}$. If $\overline{z} \subseteq \overline{y} \subseteq \overline{x}$ then $R \ltimes (S \ltimes \mathbf{t}) = (R \ltimes S) \ltimes \mathbf{t}$.*

*Proof.* This results well-know in standard relational algebra, and its proof transfers to the case of positive GMRs. □

**Lemma 2** *For every node $n \in N$ and every tuple $\mathbf{t}$ in $\rho_n$, $\textsc{enum}_{T,N}(n, \mathbf{t}, \rho)$ enumerates $\mathcal{Q}[\![T_n, N_n]\!](db) \ltimes \mathbf{t}$.*

*Proof.* Let $n \in N$ and $\mathbf{t} \in \rho_n$. We need to show that executing $\textsc{enum}_{T,N}(n, \mathbf{t}, \rho)$ outputs all (tuple, multiplicity) pairs of $\mathcal{Q}[\![T_n, N_n]\!](db) \ltimes \mathbf{t}$ exactly once. We proceed by induction on the number of nodes in $N_n$. If $N_n = \{n\}$ then $\mathcal{Q}[\![T_n, N_n]\!] = \mathcal{Q}[\![T_n, n]\!]$. Therefore, by Lemma 1, $\mathcal{Q}[\![T_n, N_n]\!](db) = \rho_n$. Since $\mathbf{t} \in \rho_n$, this implies that the only tuple in $\mathcal{Q}[\![T_n, N_n]\!](db)$ that is compatible with $\mathbf{t}$ is $\mathbf{t}$ itself. Furthermore, since $N_n = \{n\}$, $n$ must be in the frontier of $n$. Therefore, $\textsc{enum}_{T,N}(n, \mathbf{t}, \rho)$ will output precisely $\{(\mathbf{t}, \rho_n(\mathbf{t}))\}$ (Line 4), which concludes the base case.

For the inductive step we need to consider two cases depending on the number of children of $n$.

Case (1). If $n$ has a single child $c$ then necessarily $c$ is a guard of $n$, i.e., $var(n) \subseteq var(c)$. In this case, Algorithm 1 will call $\textsc{enum}_{T,N}(c, \mathbf{s}, \rho)$ for each tuple $\mathbf{s} \in \left(\rho_c \ltimes_{pred(n)} \mathbf{t}\right)$. By induction hypothesis and Lemma 1, this will correctly enumerate and output the elements of $\mathcal{Q}[\![T_c, N_c]\!](db) \ltimes \mathbf{s}$, for every $\mathbf{s}$ in $\mathcal{Q}[\![T_c, c]\!](db) \ltimes_{pred(n)} \mathbf{t}$. Note that the sets $\mathcal{Q}[\![T_c, N_c]\!](db) \ltimes \mathbf{s}$ are disjoint for different values of $\mathbf{s}$. Thus, no element is output twice. Hence, $\textsc{enum}_{T,N}(n, \mathbf{t}, \rho)$ enumerates the GMR

$$\mathcal{Q}[\![T_c, N_c]\!](db) \ltimes (\mathcal{Q}[\![T_c, c]\!](db) \ltimes_{pred(n)} \mathbf{t}). \tag{2}$$

Since $var(pred(n)) \subseteq var(c) \cup var(n) = var(c) = out(\mathcal{Q}[\![T_c, c]\!])$, we can pull out the selection:

$$(2) = \mathcal{Q}[\![T_c, N_c]\!](db) \ltimes \sigma_{pred\,n} (\mathcal{Q}[\![T_c, c]\!](db) \ltimes \mathbf{t}). \tag{3}$$

Subsequently, because $var(pred(n)) = var(c) \subseteq out(\mathcal{Q}[\![T_c, N_c]\!])$, we can pull out the selection again:

$$(3) = \sigma_{pred(n)} \left(\mathcal{Q}[\![T_c, N_c]\!](db) \ltimes (\mathcal{Q}[\![T_c, c]\!](db) \ltimes \mathbf{t})\right). \tag{4}$$

Because the variables in $\mathbf{t}$ are a subset of $var(c)$, because $var(c) \subseteq var(N_c)$, and because $\mathcal{Q}[\![T_c, N_c]\!](db)$ and $\mathcal{Q}[\![T_c, c]\!](db)$ are positive (Lemma 3(1)) we can apply Lemma 10:

$$(4) = \sigma_{pred(n)} \left((\mathcal{Q}[\![T_c, N_c]\!](db) \ltimes \mathcal{Q}[\![T_c, c]\!](db)) \ltimes \mathbf{t}\right). \tag{5}$$

Next, observe that, since $var(n_c) \subseteq var(N_c)$ as $c \in N_c$ we have

$$\mathcal{Q}[\![T_c, n_c]\!] = \pi_{var(c)} \mathcal{Q}[\![T_c]\!] \equiv \pi_{var(c)} \pi_{var(N_c)} \mathcal{Q}[\![T_c]\!]$$

$$\equiv \pi_{var(c)} \mathcal{Q}[\![T_c, N_c]\!]$$

Then, because $\mathcal{Q}[\![T_c, N_c]\!](db)$ is positive we obtain from Lemma 3(2) that

$$(5) = \sigma_{pred(n)}(\mathcal{Q}[\![T_c, N_c]\!](db) \ltimes \mathbf{t}). \tag{6}$$

Finally, because $pred(n) \subseteq var(n) \subseteq var(N_c)$ we push the selection again, and obtain

$$(6) = (\sigma_{pred(n)} \mathcal{Q}[\![T_c, N_c]\!](db)) \ltimes \mathbf{t} \tag{7}$$

$$= (\pi_{var(N_n)} \sigma_{pred(n)} \mathcal{Q}[\![T_c, N_c]\!])(db) \ltimes \mathbf{t}. \tag{8}$$

Here, the last equality is due to the fact that $var(N_n) = var(n) \cup var(N_c) = var(N_c)$, as $var(n) \subseteq var(c)$ and $c \in N_c$, which implies that projecting on $var(N_n)$ does not modify the result. The result then follows from the observation that $\mathcal{Q}[\![T_n, N_n]\!] \equiv \pi_{var(N_n)} \sigma_{pred(n)} \mathcal{Q}[\![T_c, N_c]\!]$.

Case (2). Otherwise, $n$ has two children $c_1$ and $c_2$. We assume w.l.o.g. that $c_1$ is a guard of $n$, i.e $var(n) \subseteq var(c_1)$. Since $|N_n| > 1$ and $N$ is sibling closed we have $\{c_1, c_2\} \subset N$. In this case, Algorithm 1 will first enumerate $\mathbf{t_i} \in \rho_{c_i} \ltimes_{pred(n \to c_1)} \mathbf{t}$ for $i \in \{1, 2\}$. By Lemma 1 this is equivalent to enumerate every $\mathbf{t_i}$ in $\mathcal{Q}[\![T_{c_i}, c_i]\!](db) \ltimes_{pred(n \to c_1)} \mathbf{t}$. Then, for each such $\mathbf{t_i}$ the algorithm will enumerate every pair $(\mathbf{s_i}, \mu_i)$ generated by $\text{ENUM}_{T,N}(c_i, \mathbf{t_i}, \rho)$, which by induction is the same as enumerating every $(\mathbf{s_i}, \mu_i)$ in $\mathcal{Q}[\![T_{c_i}, N_{c_i}]\!](db) \ltimes \mathbf{t_i}$. Note that the sets $\mathcal{Q}[\![T_{c_i}, N_{c_i}]\!](db) \ltimes \mathbf{t_i}$ are disjoint for distinct $\mathbf{t_i}$. Therefore, no $(\mathbf{s_i}, \mu_i)$ is generated twice. the algorithm is hence enumerating

$$\mathcal{Q}[\![T_{c_i}, N_{c_i}]\!](db) \ltimes \left(\mathcal{Q}[\![T_{c_i}, c_i]\!](db) \ltimes_{pred(n \to c_i)} \mathbf{t}\right)$$

By the same reasoning as in Case (1), this is equivalent to enumerating every $(\mathbf{s_i}, \mu_i)$ in $(\sigma_{pred(n \to c_i)} \mathcal{Q}[\![T_{c_i}]\!](db)) \ltimes \mathbf{t}$. From the connectedness property of $T$, it follows that $var(\mathcal{Q}[\![T_{c_1}]\!]) \cap var(\mathcal{Q}[\![T_{c_2}]\!]) \subseteq var(n)$. Thus, $var(\mathcal{Q}[\![T_{c_1}]\!]) \cap var(\mathcal{Q}[\![T_{c_2}]\!])$ is a subset of the variables of $\mathbf{t}$. Hence, every tuple $\mathbf{s_1}$ will be compatible with every tuple $\mathbf{s_2}$, and therefore enumeration of every pair $(\mathbf{s_1} \cup \mathbf{s_2}, \mu_1 \times \mu_2)$ is the same as the enumeration of

$$\left((\sigma_{pred(n \to c_1)} \mathcal{Q}[\![T_{c_1}, N_{c_1}]\!](db)) \ltimes \mathbf{t}\right) \bowtie \left((\sigma_{pred(n \to c_2)} \mathcal{Q}[\![T_{c_2}, N_{c_2}]\!](db)) \ltimes \mathbf{t}\right). \tag{9}$$

The semijoin with $\mathbf{t}$ factors out of the join:

$$(9) = \left(\sigma_{pred(n \to c_1)} \mathcal{Q}[\![T_{c_1}, N_{c_1}]\!] \bowtie \sigma_{pred(n \to c_2)} \mathcal{Q}[\![T_{c_2}, N_{c_2}]\!]\right)(db) \ltimes \mathbf{t} \tag{10}$$

We can now pull out the selections and obtain

$$(10) = \left(\sigma_{pred(n \to c_1)} \sigma_{pred(n \to c_2)} (\mathcal{Q}[\![T_{c_1}, N_{c_1}]\!] \bowtie \mathcal{Q}[\![T_{c_2}, N_{c_2}]\!])(db)\right) \ltimes \mathbf{t}.$$

$$= \left(\sigma_{pred(n)} (\mathcal{Q}[\![T_{c_1}, N_{c_1}]\!] \bowtie \mathcal{Q}[\![T_{c_2}, N_{c_2}]\!])(db)\right) \ltimes \mathbf{t}.$$

$$= \left(\pi_{var(N_n)} \sigma_{pred(n)} (\mathcal{Q}[\![T_{c_1}, N_{c_1}]\!] \bowtie \mathcal{Q}[\![T_{c_2}, N_{c_2}]\!])(db)\right) \ltimes \mathbf{t}$$

Here, the last equality is due to the fact that $var(N_n) = var(n) \cup var(N_{c_1}) \cup var(N_{c_2}) = var(N_{c_1}) \cup var(N_{c_2})$, as $var(n) \subseteq var(c_1) \subseteq var(N_{c_1})$. This implies that

$$var(N_n) = out(\mathcal{Q}[\![T_{c_1}, N_{c_1}]\!]) \cup out(\mathcal{Q}[\![T_{c_2}, N_{c_2}]\!])$$

Hence, projecting the join result on $var(N_n)$ does not modify the result. The result then follows from the observation that $\mathcal{Q}[\![T_n, N_n]\!] \equiv \pi_{var(N_n)} \sigma_{pred(n)} (\mathcal{Q}[\![T_{c_1}, N_{c_2}]\!] \bowtie \mathcal{Q}[\![T_{c_2}, N_{c_2}]\!])$. □

**Proposition 4** *Assume that all join indexes in the $(T, N)$-rep have access time $g$, and that all indexes (join and enumeration) have update time $h$, where $g$ and $h$ are monotone functions. Further assume that, during the entire execution of* UPDATE*, $K$ and $U$ bound the size of $\rho_n$, resp. $\Delta_n$, for all $n$. Then, $\text{UPDATE}_{T,N}(\rho, u)$ runs in time $\mathcal{O}(|T| \cdot (U + h(K, U) + g(K, U)))$.*

*Proof.* First note that the initialization of $\Delta_n$ in line 15 can be done in $\mathcal{O}(U)$ time (by copying $u_{r(\overline{x})}$ to $\Delta_n$ tuple by tuple) and the initialization of $\Delta_n$ in line 17 in $\mathcal{O}(1)$ time. Therefore, lines 14–17 run in $\mathcal{O}(|T| \cdot U)$ time, which falls within the claimed bounds. We next show that the for-loop of line 18–23 also runs within the claimed bounds. Since the body of this for-loop is executed $|T|$ times, it suffices to show that each of the lines 19–23 run in time $\mathcal{O}(U + h(K, U) + g(K, U))$. Since $|\Delta_n| \leq U$ by assumption, the statement $\rho_n += \Delta_n$ of line 19 can be executed in $\mathcal{O}(U)$ time by iterating over the tuples $\mathbf{t} \in \Delta_n$, and updating $\rho_n(\mathbf{t})$ for each such tuple. (Recall that multiplicity lookup and modification in a GMR are $\mathcal{O}(1)$ operations). The indexes associated to $\rho_n$ (if any) are updated in time $h(K, U)$. Therefore, the total time require to execute line 19 is $\mathcal{O}(U + h(K, U))$. We next bound the complexity of line 21. Computing $\pi_{var(p)}(\rho_m \bowtie_{pred\, p} \Delta_n)$ using the join index on $\rho_m$ takes $\mathcal{O}(g(K, U))$ time. Furthermore, the number of tuples in $\pi_{var(p)}(\rho_m \bowtie_{pred\, p} \Delta_n)$ can be at most $2U$. This is because $|\Delta_p| \leq U$ at any time during the execution. In the worst case, therefore, $\pi_{var(p)}(\rho_m \bowtie_{pred\, p} \Delta_n)$ can at most delete the tuples already present in $\Delta_p$ (which requires $U$ tuples), and subsequently insert $U$ new tuples (requiring another $U$ tuples), for at most $2U$ tuples in total. For each of the $2U$ resulting tuples we update $\Delta_p$ accordingly in $\mathcal{O}(1)$ time. The total time to execute line 21 is hence $\mathcal{O}(2 \cdot U + g(K, U))$. Finally, using similar reasoning, the complexity of line 23 can be shown to be $\mathcal{O}(U)$. □

## B Proofs of Section 5.1

### B.1 Proof of Proposition 7

Because no infinite sequences of reduction steps are possible, it suffices to demonstrate local confluence:

**Proposition 14** *If $\mathcal{H} \rightsquigarrow \mathcal{I}_1$ and $\mathcal{H} \rightsquigarrow \mathcal{I}_2$ then there exists $\mathcal{J}$ such that both $\mathcal{I}_1 \rightsquigarrow^* \mathcal{J}$ and $\mathcal{I}_2 \rightsquigarrow^* \mathcal{J}$.*

Indeed, it is a standard result in the theory of rewriting systems that confluence (Lemma 7) and local confluence (lemma 14) coincide when infinite sequences of reductions steps are impossible [5].

Before proving Lemma 14, we observe that the property of being isolated or being a conditional subset is preserved under reductions, in the following sense.

**Lemma 11** *Assume that $\mathcal{H} \rightsquigarrow \mathcal{I}$. Then $pred(\mathcal{I}) \subseteq pred(\mathcal{H})$ and for every hyperedge $e$ we have $ext_{\mathcal{I}}(e) \subseteq ext_{\mathcal{H}}(e)$, $jv_{\mathcal{I}}(e) \subseteq jv_{\mathcal{H}}(e)$, and $isol_{\mathcal{H}}(e) \subseteq isol_{\mathcal{I}}(e)$. Furthermore, if $e \sqsubseteq_{\mathcal{H}} f$ then also $e \sqsubseteq_{\mathcal{I}} f$.*

*Proof.* First observe that $pred(\mathcal{I}) \subseteq pred(\mathcal{H})$, since reduction operators only remove predicates. This implies that $ext_{\mathcal{I}}(e) \subseteq ext_{\mathcal{H}}(e)$ for every hyperedge $e$. Furthermore, because reduction operators only remove hyperedges and never add them, it is easy to see that $jv_{\mathcal{H}}(e) \subseteq jv_{\mathcal{I}}(e)$. Hence, if $x \in isol_{\mathcal{H}}(e)$ then $x \notin jv_{\mathcal{H}}(e) \supseteq jv_{\mathcal{I}}(e)$ and $x \notin var(pred(\mathcal{H})) \supseteq var(pred(\mathcal{I}))$. Therefore, $x \in isol_{\mathcal{I}}(e)$. As such, $isol_{\mathcal{I}}(e) \subseteq isol_{\mathcal{H}}(e)$.

Next, assume that $e \sqsubseteq_{\mathcal{H}} f$. We need to show that $jv_{\mathcal{I}}(e) \subseteq f$ and $ext_{\mathcal{I}}(e \setminus f) \subseteq f$. The first condition follows since $jv_{\mathcal{I}}(e) \subseteq jv_{\mathcal{H}}(e) \subseteq f$ where the last inclusion is due to $e \sqsubseteq_{\mathcal{H}} f$. The second also follows since $ext_{\mathcal{I}}(e \setminus f) \subseteq ext_{\mathcal{H}}(e \setminus f) \subseteq f$ where the last inclusion is due to $e \sqsubseteq_{\mathcal{H}} f$. □

*Proof of Proposition 14.* If $\mathcal{I}_1 = \mathcal{I}_2$ then it suffices to take $\mathcal{J} = \mathcal{I}_1 = \mathcal{I}_2$. Therefore, assume in the following that $\mathcal{I}_1 \neq \mathcal{I}_2$. Then, necessarily $\mathcal{I}_1$ and $\mathcal{I}_2$ are obtained by applying two different reduction operations on $\mathcal{H}$. We make a case analysis on the types of reductions applied.

*(1) Case (ISO, ISO)*: assume that $\mathcal{I}_1$ is obtained by removing the non-empty set $X_1 \subseteq isol_{\mathcal{H}}(e_1)$ from hyperedge $e_1$, while $\mathcal{I}_2$ is obtained by removing non-empty $X_2 \subseteq isol_{\mathcal{H}}(e_2)$ from $e_2$ with $X_1 \neq X_2$. There are two possibilities.

(1a) $e_1 \neq e_2$. Then $e_2$ is still a hyperedge in $\mathcal{I}_2$ and $e_1$ is still a hyperedge in $\mathcal{I}_1$. By Lemma 11, $isol_{\mathcal{H}}(e_1) \subseteq isol_{\mathcal{I}_2}(e_1)$ and $isol_{\mathcal{H}}(e_2) \subseteq isol_{\mathcal{I}_1}(e_2)$. Therefore, we can still remove $X_2$ from $\mathcal{I}_1$ by means of rule ISO, and similarly remove $X_1$ from $\mathcal{I}_2$. Let $\mathcal{J}_1$ (resp. $\mathcal{J}_2$) be the result of removing $X_2$ from $\mathcal{I}_1$ (resp. $\mathcal{I}_2$). Then $\mathcal{J}_1 = \mathcal{J}_2$ (and hence equals triplet $\mathcal{J}$):

$$\begin{aligned} hyp(\mathcal{J}_1) &= hyp(\mathcal{H}) \setminus \{e_1, e_2\} \cup \{e_1 \setminus X_1 \mid e_1 \setminus X_1 \neq \emptyset\} \\ &\quad \cup \{e_2 \setminus X_2 \mid e_2 \setminus X_2 \neq \emptyset\} \\ &= hyp(\mathcal{J}_2) \\ pred(\mathcal{J}_1) &= pred(\mathcal{H}) = pred(\mathcal{J}_2) \end{aligned}$$

(1b) $e_1 = e_2$. We show that $X_2 \setminus X_1 \subseteq isol_{\mathcal{I}_1}(e_1 \setminus X_1)$ and similarly $X_1 \setminus X_2 \subseteq isol_{\mathcal{I}_1}(e_2 \setminus X_1)$. This suffices because we can then apply ISO to remove $X_2 \setminus X_1$ from $\mathcal{I}_1$ and $X_1 \setminus X_2$ from $\mathcal{I}_2$. In both cases, we reach the same triplet as removing $X_1 \cup X_2 \subseteq isol_{\mathcal{H}}(e_1)$ from $\mathcal{H}$.[7]

To see that $X_2 \setminus X_1 \subseteq isol_{\mathcal{I}_1}(e_1 \setminus X_1)$, let $x \in X_2 \setminus X_1$. We need to show $x \notin jv_{\mathcal{I}_1}(e_1 \setminus X_1)$ and $x \notin var(pred(\mathcal{I}_1))$. Because $x \in X_2 \subseteq isol_{\mathcal{H}}(e_1)$ we know $x \notin jv_{\mathcal{H}}(e_1)$. Then, since $x \notin X_1$, also $x \notin jv_{\mathcal{H}}(e_1 \setminus X_1)$. By Lemma 11, $jv_{\mathcal{I}_1}(e_1 \setminus X_1) \subseteq jv_{\mathcal{H}}(e_1 \setminus X_1)$. Therefore, $x \notin jv_{\mathcal{I}_1}(e_1 \setminus X_1)$. Furthermore, because $x \in isol_{\mathcal{H}}(e_1)$ we know $x \notin var(pred(\mathcal{H}))$. Since $var(pred(\mathcal{I}_1)) \subseteq var(pred(\mathcal{H}))$ by Lemma 11, also $x$ *not* $\in var(pred(\mathcal{I}_1))$.

$X_1 \setminus X_2 \subseteq isol_{\mathcal{I}_1}(e_2 \setminus X_1)$ is shown similarly.

*(2) Case (CSE, CSE)*: assume that $\mathcal{I}_1$ is obtained by removing hyperedge $e_1$ because it is a conditional subset of hyperedge $f_1$, while $\mathcal{I}_2$ is obtained by removing $e_2$, conditional subset of $f_2$. Since $\mathcal{I}_1 \neq \mathcal{I}_2$ it must be $e_1 \neq e_2$. We need to further distinguish the following cases.

(2a) $e_1 \neq f_2$ and $e_2 \neq f_1$. In this case, $e_2$ and $f_2$ remain hyperedges in $\mathcal{I}_1$ while $e_1$ and $f_1$ remain hyperedges in $\mathcal{I}_2$. Then, by Lemma 11, $e_2 \sqsubseteq_{\mathcal{I}_1} f_2$ and $e_1 \sqsubseteq_{\mathcal{I}_2} f_2$. Let $\mathcal{J}_1$ (resp. $\mathcal{J}_2$) be the triplet obtained by removing $e_2$ from $\mathcal{I}_1$ (resp. $e_1$ from $\mathcal{I}_2$). Then $\mathcal{J}_1 = \mathcal{J}_2$ since clearly $out(\mathcal{J}_1) = out(\mathcal{J}_2)$ and

$$\begin{aligned} hyp(\mathcal{J}_1) &= hyp(\mathcal{H}) \setminus \{e_1, e_2\} = hyp(\mathcal{J}_2) \\ pred(\mathcal{J}_1) &= \{\theta \in pred(\mathcal{H}) \mid var(\theta) \cap (e_1 \setminus f_1) = \emptyset, \\ &\qquad var(\theta) \cap (e_2 \setminus f_2) = \emptyset\} \\ &= pred(\mathcal{J}_2) \end{aligned}$$

From this the result follows by taking $\mathcal{J} = \mathcal{J}_1 = \mathcal{J}_2$.

(2b) $e_1 \neq f_2$ but $e_2 = f_1$. Then $e_1 \sqsubseteq_{\mathcal{H}} e_2$ and $e_2 \sqsubseteq_{\mathcal{H}} f_2$ with $f_2 \neq e_1$. It suffices to show that $e_1 \sqsubseteq_{\mathcal{H}} f_2$ and $e_1 \setminus f_2 = e_1 \setminus f_1$, because then (CSE) due to $e_1 \sqsubseteq_{\mathcal{H}} f_1$ has the same effect as CSE on $e_1 \sqsubseteq_{\mathcal{H}} f_2$, and we can apply the reasoning of case (2a) because $e_1 \neq f_2$ and $e_2 \neq f_2$.

We first show $e_1 \setminus f_2 = e_1 \setminus f_1$. Let $x \in e_1 \setminus f_2$ and suppose for the purpose of contradiction that that $x \in e_2 = f_1$. Then, since $e_1 \neq e_2$, $x \in jv(e_2) \subseteq f_2$ where the last inclusion is due to $e_2 \sqsubseteq_{\mathcal{H}} f_2$. Hence, $e_1 \setminus f_2 \subseteq e_1 \setminus f_1$. Conversely, let $x \in e_1 \setminus f_1$. Since $f_1 = e_2$, $x \notin e_2$. Suppose for the purpose of contradiction that $x \in f_2$. Because $e_1 \neq f_2$, $x \in jv_{\mathcal{H}}(e_1) \subseteq e_2$ where the last inclusion is due to $e_1 \sqsubseteq_{\mathcal{H}} e_2$. Therefore, $e_2 \setminus f_1 = e_1 \setminus f_2$.

To show that $e_1 \sqsubseteq_{\mathcal{H}} f_2$, let $x \in jv_{\mathcal{H}}(e_1)$. Because $e_1 \sqsubseteq_{\mathcal{H}} e_2$, $x \in e_2$. Because $x$ occurs in two distinct hyperedges in $\mathcal{H}$, also $x \in jv_{\mathcal{H}}(e_2)$. Then, because $e_2 \sqsubseteq_{\mathcal{H}} f_2$, $x \in f_2$. Hence $jv_{\mathcal{H}}(e_1) \subseteq f_2$. It remains to show $ext_{\mathcal{H}}(e_1 \setminus f_2) \subseteq f_2$. To this end, let $x \in ext_{\mathcal{H}}(e_1 \setminus f_2)$ and suppose for the purpose of contradiction that $x \notin f_2$. By definition of *ext* there exists $\theta \in pred(\mathcal{H})$ and $y \in var(\theta) \cap (e_1 \setminus f_2)$ such that $x \in var(\theta) \setminus (e_1 \setminus f_2)$. In particular, $y \notin f_2$. Since $e_1 \setminus f_2 = e_1 \setminus e_2$, $y \in var(\theta) \cap (e_1 \setminus e_2)$ and $x \in var(\theta) \setminus (e_1 \setminus e_2)$. Thus, $x \in ext_{\mathcal{H}}(e_1 \setminus e_2)$. Then, since $e_1 \sqsubseteq_{\mathcal{H}} e_2$, $x \in e_2$. Thus, $x \in e_2 \setminus f_2$ since $x \notin f_2$. Hence $x \in var(\theta) \cap (e_2 \setminus f_2)$. Furthermore, since $y \notin e_2$ also $y \notin e_2 \setminus f_2$. Hence, $y \in var(\theta) \setminus (e_2 \setminus f_2)$. But then $\theta$ shows that $y \in ext_{\mathcal{H}}(e_2 \setminus f_2)$. Then, by because $e_2 \sqsubseteq_{\mathcal{H}} f_2$, also $y \in f_2$ which yields the desired contradiction.

(2c) $e_1 = f_2$ but $e_2 \neq f_1$. Similar to case (2b).

(2d) $e_1 = f_2$ and $e_2 = f_1$. Then $e_1 \sqsubseteq_{\mathcal{H}} e_2$ and $e_2 \sqsubseteq_{\mathcal{H}} e_1$ and $e_1 \neq e_2$. Let $\mathcal{K}_1$ (resp. $\mathcal{K}_2$) be the triplet obtained by applying (FLT) to remove all $\theta \in pred(\mathcal{I}_1)$ (resp. $\theta \in pred(\mathcal{I}_2)$ for which $var(\theta) \subseteq var(e_2)$ (resp. $(var(\theta) \subseteq var(e_2)$. Furthermore, let $\mathcal{J}_1$ (resp. $\mathcal{J}_2$) be the triplet obtained by applying ISO to removing $isol_{\mathcal{I}_1}(e_2)$ from $\mathcal{K}_1$ (resp. removing $isol_{\mathcal{I}_2}(e_1)$ from $\mathcal{K}_2$). Here, we take $\mathcal{J}_1 = \mathcal{K}_1$ if $isol_{\mathcal{K}_1}(e_2)$ is empty (and similarly for $\mathcal{J}_2$). Then clearly $\mathcal{H} \rightsquigarrow \mathcal{I}_1 \rightsquigarrow^* \mathcal{K}_1 \rightsquigarrow^* J_1$ and $\mathcal{H} \rightsquigarrow \mathcal{I}_2 \rightsquigarrow^* \mathcal{K}_2 \rightsquigarrow^* \mathcal{J}_2$. The result then follows by showing that $\mathcal{J}_1 = \mathcal{J}_2$. Towards this end, first observe that $out(\mathcal{J}_1) = out(\mathcal{K}_1) = out(\mathcal{I}_1) = out(\mathcal{H}) = out(\mathcal{I}_2) = out(\mathcal{K}_2) = out(\mathcal{J}_2)$. Next, we show that $pred(\mathcal{J}_1) = pred(\mathcal{J}_2)$. We first observe that $pred(\mathcal{J}_1) = pred(\mathcal{K}_1)$ and $pred(\mathcal{J}_2) = pred(\mathcal{K}_2)$ since the ISO operation does not remove predicates. Then observe that

$$\begin{aligned} pred(\mathcal{K}_1) &= \{\theta \in pred(\mathcal{I}_1) \mid var(\theta) \not\subseteq var(e_2)\} \\ &= \{\theta \in pred(\mathcal{H}) \mid var(\theta) \cap (e_1 \setminus e_2) = \emptyset \text{ and} \\ &\qquad var(\theta) \not\subseteq e_2\}, \\ pred(\mathcal{K}_2) &= \{\theta \in pred(\mathcal{I}_2) \mid var(\theta) \not\subseteq e_1\} \\ &= \{\theta \in pred(\mathcal{H}) \mid var(\theta) \cap (e_2 \setminus e_1) = \emptyset \text{ and} \\ &\qquad var(\theta) \not\subseteq e_1\}. \end{aligned}$$

We only show the reasoning for $pred(\mathcal{K}_1) \subseteq pred(\mathcal{K}_2)$, the other direction being similar. Let $\theta \in pred(\mathcal{K}_1)$. Then $var(\theta \cap (e_1 \setminus e_2) = \emptyset$ and $var(\theta) \not\subseteq e_2$. Since $var(\theta) \not\subseteq e_2$ there exists $y \in var(\theta) \setminus e_2$. Then, because $var(\theta) \cap (e_1 \setminus e_2) = \emptyset$, $y \notin e_1$. Thus, $var(\theta) \not\subseteq e_1$. Now, suppose for the purpose of obtaining a contradiction, that $var(\theta) \cap (e_2 \setminus e_1) \neq \emptyset$. Then take $z \in var(\theta) \cap (e_2 \setminus e_1)$. But then $y \in ext_{\mathcal{H}}(e_2 \setminus e_1)$. Hence, $y \in e_1$ because $e_2 \sqsubseteq_{\mathcal{H}} e_1$, which yields the desired contradiction with $y \notin e_2$. Therefore, $var(\theta) \cap (e_2 \setminus e_1) = \emptyset$, as desired. Hence $\theta \in pred(\mathcal{K}_2)$.

It remains to show that $hyp(\mathcal{J}_1) = hyp(\mathcal{J}_2)$. To this end, first observe

$$\begin{aligned} hyp(\mathcal{J}_1) &= hyp(\mathcal{K}_1) \setminus \{e_2\} \cup \{e_2 \setminus isol_{\mathcal{K}_1}(e_2)\}, \\ &= hyp(\mathcal{H}) \setminus \{e_1\} \setminus \{e_2\} \cup \{e_2 \setminus isol_{\mathcal{K}_1}(e_2)\}, \\ hyp(\mathcal{J}_2) &= hyp(\mathcal{K}_2) \setminus \{e_1\} \cup \{e_1 \setminus isol_{\mathcal{K}_2}(e_1)\} \\ &= hyp(\mathcal{H}) \setminus \{e_2\} \setminus \{e_1\} \cup \{e_1 \setminus isol_{\mathcal{K}_2}(e_1)\}. \end{aligned}$$

Clearly, $hyp(\mathcal{J}_1) = hyp(\mathcal{J}_2)$ if $e_2 \setminus isol_{\mathcal{K}_1}(e_2) = e_1 \setminus isol_{\mathcal{K}_2}(e_1)$.

[7] Should $X_2 \setminus X_1$ be empty, we don't actually need to do anything on $\mathcal{I}_1$: $X_1 \cup X_2$ is already removed from it. A similar remark holds for $\mathcal{I}_2$ when $X_1 \setminus X_2$ is empty.

We only show $e_2 \setminus isol_{\mathcal{K}_1}(e_2) \subseteq e_1 \setminus isol_{\mathcal{K}_2}(e_1)$, the other inclusion being similar. Let $x \in e_2 \setminus isol_{\mathcal{K}_1}(e_2)$. Since $x \notin isol_{\mathcal{K}_1}(e_2)$ one of the following hold.

- $x \in out(\mathcal{K}_1)$. But then, $x \in out(\mathcal{K}_1) = out(\mathcal{I}_1) = out(\mathcal{H}) = out(\mathcal{I}_2) = out(\mathcal{K}_2)$. In particular, $x$ is an equijoin variable in $\mathcal{H}$ and $\mathcal{K}_\in$. Then $x \in jv_{\mathcal{H}}(e_2) \subseteq e_1$ because $e_2 \sqsubseteq_{\mathcal{H}} e_1$. From this and the fact that $x$ remains an equijoin variable in $\mathcal{K}_2$, we obtain $x \in e_1 \setminus isol_{\mathcal{K}_2}(e_1)$.
- $x$ occurs in $e_2$ and in some hyperedge $g$ in $\mathcal{K}_1$ with $g \neq e_2$. Since $e_1$ is not in $\mathcal{K}_1$ also $g \neq e_1$. Since every hyperedge in $\mathcal{K}_1$ is in $\mathcal{I}_1$ and every hyperedge in $\mathcal{I}_1$ is in $\mathcal{H}$, also $g$ is in $\mathcal{H}$. But then, $x$ occurs in two distinct hyperedges in $\mathcal{H}$, namely $e_2$ and $g$, and hence $x \in jv_{\mathcal{H}}(e_2) \subseteq e_1$ because $e_2 \sqsubseteq_{\mathcal{H}} e_1$. However, because $x$ also occurs in $g$ which must also be in $\mathcal{I}_2$ and therefore also in $\mathcal{K}_2$, $x$ also occurs in two distinct hyperedges in $\mathcal{K}_2$, namely $e_1$ and $g$. Therefore, $x \in jv_{\mathcal{I}_2}(e_1)$ and hence $x \in e_1 \setminus isol_{\mathcal{I}_2}(e_1)$, as desired.
- $x \in var(pred(\mathcal{K}_1))$. Then there exists $\theta \in pred(\mathcal{K}_1)$ such that $x \in var(\theta)$. Since $pred(\mathcal{K}_1) = pred(\mathcal{K}_2)$, $\theta \in pred(\mathcal{K}_2)$. As such, $\theta \in pred(\mathcal{H})$, $var(\theta) \cap (e_2 \setminus e_1) = \emptyset$, and $var(\theta) \not\subseteq e_1$. But then, since $x \in var(\theta)$; $x \in e_2$; and $var(\theta) \cap (e_2 \setminus e_1) = \emptyset$, it must be the case that $x \in e_1$. As such, $x \in e_1$ and $x \in var(\mathcal{K}_2)$. Hence $x \in e_1 \setminus isol_{\mathcal{K}_2}(e_1)$.

*(3) Case (ISO, CSE)*: assume that $\mathcal{I}_1$ is obtained by removing the non-empty set of isolated variables $X_1 \subseteq isol_{\mathcal{H}}(e_1)$ from $e_1$, while $\mathcal{I}_2$ is obtained by removing hyperedge $e_2$, conditional subset of hyperedge $f_2$. We may assume w.l.o.g. that $e_1 \neq isol_{\mathcal{H}}(e_1)$: if $e_1 = isol_{\mathcal{H}}(e_1)$ then the ISO operation removes the complete hyperedge $e_1$. However, because no predicate in $\mathcal{H}$ shares any variable with $e_1$, it is readily verified that $e_1 \sqsubseteq_{\mathcal{H}} e_2$ and thus the removal of $e_1$ can also be seen as an application of CSE on $e_1$[8], and we are hence back in case (2).

Now reason as follows. Because $e_2 \sqsubseteq_{\mathcal{H}} f_2$ and because isolated variables of $e_1$ occur in no other hyperedge in $\mathcal{H}$, it must be the case that $e_2 \cap X_1 = \emptyset$. In particular, $e_1$ and $e_2$ must hence be distinct. Therefore, $e_1 \in hyp(\mathcal{I}_2)$ and $e_2 \in hyp(\mathcal{I}_1)$. By Lemma 11, we can apply ISO on $\mathcal{I}_2$ to remove $X_1$ from $e_1$. It then suffices to show that $e_2$ remains a conditional subset of some hyperedge $f_2'$ in $\mathcal{I}_1$ with $e_2 \setminus f_2 = e_2 \setminus f_2'$. Indeed, we can then use ECQ to remove $e_2$ from $hyp(\mathcal{I}_1)$ as well as predicates $\theta$ with $var(\theta) \cap (e_2 \setminus f_2) \neq \emptyset$ from $pred(\mathcal{I}_1)$. This clearly yields the same triplet as the one obtained by removing $X_1$ from $e_1$ in $\mathcal{I}_2$. We need to distinguish two cases.

(3a) $f_2 \neq e_1$. Then $f_2 \in hyp(\mathcal{I}_1)$ and hence $e_2 \sqsubseteq_{\mathcal{I}_1} f_2$ by Lemma 11. We hence take $f_2' = f_2$.

(3b) $f_2 = e_1$. Then we take $f_2' = e_1 \setminus X$. Since $e_1 \neq isol_{\mathcal{H}}(e_1)$ it follows that $e_1 \setminus X_1 \neq \emptyset$. Therefore, $f_2' = e_1 \setminus X_1 \in hyp(\mathcal{I}_1)$. Furthermore, since $X \subseteq isol_{\mathcal{H}}(e_1)$, no variable in $X$ is in any other hyperedge in $\mathcal{H}$. In particular $X \cap e_2 = \emptyset$. Therefore, $e_2 \setminus f_2' = e_2 \setminus (e_1 \setminus X) = (e_2 \setminus e_1) \cup (e_2 \cap X) = e_2 \setminus e_1 \setminus e_1 = e_2 \setminus f_2$. It remains to show that $e_2 \sqsubseteq_{\mathcal{I}_1} e_1 \setminus X_1$.

- $jv_{\mathcal{I}_1}(e_2) \subseteq e_1 \setminus X_1$. Let $x \in jv_{\mathcal{I}_1}(e_2)$. By Lemma 11, $x \in jv_{\mathcal{I}_1}(e_2) \subseteq jv_{\mathcal{H}}(e_2) \subseteq e_1$ where the last inclusion is due to $e_2 \sqsubseteq_{\mathcal{H}} e_1$. In particular, $x$ is an equijoin variable in $\mathcal{H}$. But then it cannot be an isolated variable in any hyperedge. Therefore, $x \notin X_1$.
- $ext_{\mathcal{I}_1}(e_2 \setminus e_1) \subseteq e_1 \setminus X$. Let $x \in ext_{\mathcal{I}_1}(e_2 \setminus e_1)$. Then $x \in ext_{\mathcal{I}_1}(e_2 \setminus e_1) \subseteq ext_{\mathcal{H}}(e_2 \setminus e_1) \subseteq e_1$ where the first inclusion is by Lemma 11 and the second by $e_2 \sqsubseteq_{\mathcal{H}} e_1$. Then, because $x \in ext_{\mathcal{H}}(e_2 \setminus e_1)$ it follows from the definition of $ext$, that $x$ occurs in some predicate in $pred(\mathcal{H})$. However, $X$ is disjoint with $var(pred(\mathcal{H}))$ since it consist only of isolated variables. Therefore, $x \notin X$.

*(4): Case (ISO, FLT)* Assume that $\mathcal{I}_1$ is obtained by removing the non-empty set $X_1 \subseteq isol_{\mathcal{H}}(e_1)$ from hyperedge $e_1$, while $\mathcal{I}_2$ is obtained by removing all predicates in the non-empty set $\Theta \subseteq pred(\mathcal{H})$ with $var(\Theta) \subseteq e_2$ for some hyperedge $e_2$ in $hyp(\mathcal{H})$. Observe that $e_1 \in hyp(\mathcal{I}_2)$. By Lemma 11, $X \subseteq isol_{\mathcal{H}}(e_1) \subseteq isol_{\mathcal{I}_2}(e_1)$. Therefore, we may apply reduction operation (ISO) on $\mathcal{I}_2$ to remove $X_1$ from $e_1$. We will now show that, similarly, we may still apply (FLT) on $\mathcal{I}_1$ to remove all predicates in $\Theta$ from $pred(\mathcal{I}_1) = pred(\mathcal{H})$. The two operations hence commute, and clearly the resulting triplets in both cases is the same. We distinguish two possibilities. (i) $e_1 \neq e_2$. Then $e_2 \in \mathcal{I}_1$ and, $var(\Theta) \subseteq e_2$ and, since (ISO) does not remove predicates, $\Theta \subseteq pred(\mathcal{H}) = pred(\mathcal{I}_1)$. As such the (FLT) operation indeed applies to remove all predicates in $\Theta$ from $pred(\mathcal{I}_1)$. (ii) $e_1 = e_2$. Then, since $X \subseteq isol_{\mathcal{H}}(e_1)$ and isolated variables do no occur in any predicate, $X \cap var(\Theta) = \emptyset$. Then, since $var(\Theta) \subseteq e_2 = e_1$, it follows that also $var(\Theta) \subseteq e_1 \setminus X$. In particular, since we disallow nullary predicates and $\Theta$ is non-empty, $e_1 \setminus X \neq \emptyset$. Thus, $e_1 \setminus X \in hyp(\mathcal{I}_1)$ and hence operation (FLT) applies indeed applies to remove all predicates in $\Theta$ from $pred(\mathcal{I}_1)$

*(5) Case (CSE, FLT):* assume that $\mathcal{I}_1$ is obtained by removing hyperedge $e_1$, conditional subset of $e_2$ in $\mathcal{H}$, while $\mathcal{I}_2$ is obtained by removing all predicates in the non-empty set $\Theta \subseteq pred(\mathcal{H})$ with $var(\Theta) \subseteq e_3$ for some hyperedge $e_3 \in hyp(\mathcal{H})$. Since the (FLT) operation does not remove any hyperedges, $e_1$ and $e_2$ are in $hyp(\mathcal{I}_2)$. Then, since $e_1 \sqsubseteq_{\mathcal{H}} e_2$ also $e_1 \sqsubseteq_{\mathcal{I}_2} e_2$ by Lemma 11. Therefore, we may apply reduction operation (CSE) on $\mathcal{I}_2$ to remove $e_1$ from $hyp(\mathcal{I}_2)$ as well as all predicates $\theta \in pred(\mathcal{I}_2)$ for which $var(\theta) \cap (e_1 \setminus e_2) \neq \emptyset$. Let $\mathcal{J}_2$ be the triplet resulting from this operation. We will show that, similarly, we may apply (FLT) on $\mathcal{I}_1$ to remove all predicates in $\Theta \cap pred(\mathcal{I}_1)$ from $pred(\mathcal{I}_1)$, resulting in a triplet $\mathcal{J}_1$. Observe that necessarily, $\mathcal{J}_1 = \mathcal{J}_2$ (and hence they form the triplet $\mathcal{J}$). Indeed, $out(\mathcal{J}_1) = out(\mathcal{I}_1) = out(\mathcal{H}) = out(\mathcal{I}_2) = out(\mathcal{J}_2)$ since reduction operations never modify output variables. Moreover,

$$\begin{aligned} hyp(\mathcal{J}_1) &= hyp(\mathcal{I}_1) \\ &= hyp(\mathcal{H}) \setminus \{e_1\} \\ &= hyp(\mathcal{I}_2) \setminus \{e_1\} \\ &= hyp(\mathcal{J}_2) \end{aligned}$$

where the first and third equality is due to fact that (FLT) does not modify the hypergraph of the triplet it operates on. Finally, observe

$$\begin{aligned} pred(\mathcal{J}_1) &= pred(\mathcal{I}_1) \setminus (\Theta \cap pred(\mathcal{I}_1)) \\ &= pred(\mathcal{I}_1) \setminus \Theta \\ &= \{\theta \in pred(\mathcal{H}) \mid var(\theta) \cap (e_1 \setminus e_2) = \emptyset\} \setminus \Theta \\ &= \{\theta \in pred(\mathcal{H}) \setminus \Theta \mid var(\theta) \cap (e_1 \setminus e_2) = \emptyset\} \\ &= \{\theta \in pred(\mathcal{I}_2) \mid var(\theta) \cap (e_1 \setminus e_2) = \emptyset\} \\ &= pred(\mathcal{J}_2) \end{aligned}$$

It remains to show that we may apply (FLT) on $\mathcal{I}_1$ to remove all predicates in $\Theta \cap pred(\mathcal{I}_1)$, resulting in a triplet $\mathcal{J}_1$. There are two possibilities.

- $e_3 \neq e_1$. Then $e_3 \in \mathcal{I}_1$, $\Theta \cap pred((I_1)) \subseteq pred(\mathcal{I}_1))$, and $var(\Theta \cap pred(\mathcal{I}_1)) \subseteq var(\Theta) \subseteq e_3$. Hence the (FLT) operation indeed applies to $\mathcal{I}_1$ to remove all predicates in $\Theta \cap pred(\mathcal{I}_1)$.

[8] Note that, since $e_1$ does not share variables with any predicate, the CSE operation also does not remove any predicates from $\mathcal{H}_1$, similar to the ISO operation and hence yields $\mathcal{I}_1$.

– $e_3 = e_1$. In this case we claim that for every $\theta \in \Theta \cap pred(\mathcal{I}_1)$ we have $var(\theta) \subseteq e_2$. As such, $var(\Theta \cap pred(\mathcal{I}_1)) \subseteq e_2$. Since $e_2 \in hyp(\mathcal{I}_1)$ and $\Theta \cap pred(\mathcal{I}_1) \subseteq pred(\mathcal{I}_1)$ we may hence apply (FLT) to remove all predicates in $\Theta \cap pred(\mathcal{I}_1)$ from $\mathcal{I}_1$. Concretely, let $\theta \in \Theta \cap pred(\mathcal{I}_1)$. Because, in order to obtain $\mathcal{I}_1$, (CSE) removes all predicates from $\mathcal{H}$ that share a variable with $e_1 \setminus e_2$, we have $var(\theta) \cap (e_1 \setminus e_2) = \emptyset$. Moreover, because $\theta \in \Theta$, $var(\theta) \subseteq e_1$. Hence $var(\theta) \subseteq e_2$, as desired.

The remaining cases, (CSE, ISO), (FLT, ISO), and (FLT, CSE), are symmetric to case (3), (4), and (5), respectively. □

## B.2 Proof of Proposition 9

**Proposition 9** *For every GJT pair there exists an equivalent canonical pair.*

*Proof.* Let $T$ be a GJT. The proof proceeds in three steps. *Step 1.* Let $T_1$ be the GJT obtained from $T$ by (i) removing all predicates from $T$, and (ii) creating a new root node $r$ that is labeled by $\emptyset$ and attaching the root of $T$ to it, labeled by the empty set of predicates. $T_1$ satisfies the first canonicality condition, but is not equivalent to $T$ because it has none of $T$'s predicates. Now re-add the predicates in $T$ to $T_1$ as follows. For each edge $m \to n$ in $T$ and each predicate $\theta \in pred_T(m \to n)$, if $var(\theta) \cap (var(n) \setminus var(m)) \neq \emptyset$ then add $\theta$ to $pred_{T_1}(m \to n)$. Otherwise, if $var(\theta) \cap (var(n) \setminus var(m)) = \emptyset$, do the following. First observe that, by definition of GJTs, $var(\theta) \subseteq var(n) \cup var(m)$. Because $var(\theta) \cap (var(n) \setminus var(m)) = \emptyset$ this implies $var(\theta) \subseteq var(m)$. Because we disallow nullary predicates, $var(m) \neq \emptyset$. Let $a$ be the first ancestor of $m$ in $T_1$ such that $var(\theta) \not\subseteq var(a)$. Such an ancestor exists because the root of $T_1$ is labeled $\emptyset$. Let $b$ be the child of $a$ in $T_1$. Since $a$ is the first ancestor of $m$ with $var(\theta) \not\subseteq var(a)$, $var(\theta) \subseteq var(b)$. Therefore, $var(\theta) \subseteq var(b) \cup var(a)$ and $var(\theta) \cap (var(b) \setminus var(a)) \neq \emptyset$. As such, add $\theta$ to $pred_{T_1}(a \to b)$. After having done this for all predicates in $T$, $T_1$ becomes equivalent to $T$, and satisfies canonicality conditions (1) and (3). Then take take $N_1 = N \cup \{r\}$. Clearly, $N_1$ is a connex subset of $T_1$ and $var(N) = var(N')$. Therefore, $(T_1, N_1)$ is equivalent to $(T, N)$.

*Step 2.* Let $T_2$ be obtained from $T_1$ by adding, for each leaf node $l$ in $T_1$ a new interior node $n_l$ labeled by $var(l)$ and inserting it in-between $l$ and its parent in $T_1$. I.e., if $l$ has parent $p$ in $T_1$ then we have $p \to n_l \to l$ in $T_2$ with $pred_{T_2}(p \to n_l) = pred_{T_1}(p \to n)$ and $pred_{T_2}(n_l \to l) = \emptyset$.[9] Furthermore, let $N_2$ be the connex subset of $T_2$ obtained by replacing every leaf node $l$ in $N_1$ by its newly inserted node $n_l$. Clearly, $var(N_2) = var(N_1) = var(N)$ because $var(l) = var(n_l)$ for every leaf $l$ of $T_1$. By our construction, $(T_2, N_2)$ is equivalent to $(T, N)$; $T_2$ satisfies canonicality conditions (1), (2), and (4); and $N_2$ is canonical.

*Step 3.* It remains to enforce condition (3). To this end, observe that, by the connectedness condition of GJTs, $T_2$ violates canonicality condition (3) if and only if there exist internal nodes $m$ and $n$ where $m$ is the parent of $n$ such that $var(m) = var(n)$. In this case, we call $n$ a culprit node. We will now show how to obtain an equivalent pair $(U, M)$ that removes a single culprit node; the final result is then obtained by iterating this reasoning until all culprit nodes have been removed.

The culprit removal procedure is essentially the reverse of the binarization procedure of Fig. 9. Concretely, let $n$ be a culprit node with parent $m$ and let $n_1, \ldots, n_k$ be the children of $n$ in $T_2$. Let $U$ be the GJT obtained from $T_2$ by removing $n$ and attaching all children $n_i$ of $n$ as children to $m$ with edge label $pred_U(m \to n_i) = pred_{T_2}(n \to n_i)$, for $1 \leq i \leq k$. Because $var(n) = var(m)$, the result is still a valid GJT. Moreover, because $var(n) = var(m)$ and $T_2$ satisfied condition (4), we had $pred_{T_2}(m \to n) = \emptyset$, so no predicate was lost by the removal of $n$. Finally, define $M$ as follows. If $n \in N_2$, then set $M = N_2 \setminus \{n\}$, otherwise set $M = N_2$. In the former case, since $N_2$ is connex and $n \in N_2$, $m$ must also be in $N_2$. It is hence in $M$. Therefore, in both cases, $var(N) = var(N_2) = var(M)$. Furthermore, it is straightforward to check that $M$ is a connex subset of $U$. Finally, since $N_2$ consisted only of interior nodes of $T_2$, $M$ consists only of interior nodes of $U$ and hence remains canonical. □

[9] Note that all leafs have a parent since the root of $T_1$ is an interior node labeled by $\emptyset$.

## B.3 Proof of Lemma 5

We first require a number of auxiliary results.

We first make the following observations regarding canonical GJT pairs.

**Lemma 12** *Let $(T, N)$ be a canonical GJT pair, let $n$ be a frontier node of $N$ and let $m$ be the parent of $n$ in $T$.*

1. *$x \notin var(N \setminus \{n\})$, for every $x \in var(n) \setminus var(m)$.*
2. *$hyp(T, N \setminus \{n\}) = hyp(T, N) \setminus \{var(n)\})$.*
3. *$\theta \notin pred(m \to n)$, for every $\theta \in pred(T, N \setminus \{n\})$*
4. *$pred(T, N \setminus \{n\}) = pred(T, N) \setminus pred(m \to n)$.*
5. *$pred(m \to n) = \{\theta \in pred(T, N) \mid var(\theta) \cap (var(n) \setminus var(m)) \neq \emptyset\}$.*
6. *$pred(T, N \setminus \{n\}) = \{\theta \in pred(T, N) \mid var(\theta) \cap (var(n) \setminus var(m)) = \emptyset\}$.*

*Proof.* (1) Let $x \in var(n) \setminus var(m)$ and let $c$ be a node in $N \setminus \{n\}$. Clearly the unique undirected path between $c$ and $n$ in $T$ must pass through $m$. Because $x \notin var(m)$ it follows from the connectedness condition of GJTs that also $x \notin var(c)$. As such, $x \notin var(N \setminus \{n\})$.

(2) The $\supseteq$ direction is trivial. For the $\subseteq$ direction, assume that $m \in N \setminus \{n\}$ with $var(m) \neq \emptyset$. Then clearly $m \in N$ and hence $var(m) \in hyp(T, N)$. Furthermore, because $N$ is canonical, both $m$ and $n$ are interior nodes in $T$. Then, because $T$ is canonical and $m \neq n$ we have $var(m) \neq var(n)$. Therefore, $var(m) \in hyp(T, N) \setminus \{var(n)\}$.

(3) Let $\theta \in pred(T, N \setminus n)$. Then $\theta$ occurs on the edge between two nodes in $N \setminus n$, say $m' \to n'$. By definition of GJTs, $var(\theta) \subseteq var(n') \cup var(m') \subseteq var(N \setminus \{n\})$. Now suppose for the purpose of contradiction that also $\theta \in pred(m \to n)$. Because $T$ is nice, there is some $x \in var(\theta) \cap (var(n) \setminus var(m)) \neq \emptyset$. Hence, by (1), $x \notin var(N \setminus \{n\})$, which contradicts $var(\theta) \subseteq var(N \setminus \{n\})$.

(4) Clearly, $pred(T, N) \setminus pred(m \to n) \subseteq pred(T, N \setminus \{n\})$. The converse inclusion follows from (3).

(5) The $\subseteq$ direction follows from the fact that $m$ and $n$ are in $N$, and $T$ is nice. To also see $\supseteq$, let $\theta \in pred(T, N)$ with $var(\theta) \cap (var(n) \setminus var(m)) \neq \emptyset$. There exists $x \in var(\theta) \cap (var(n) \setminus var(m))$. By (1), $x \notin var(N \setminus \{n\})$. Therefore, $\theta$ cannot occur between edges in $N \setminus \{n\}$ in $T$. Since it nevertheless occurs in $pred(T, N)$, it must hence occur in $pred(m \to n)$.

(6) Follows directly from (4) and (5). □

**Lemma 13** *Let $(T, N)$ be a canonical GJT pair, let $n$ be a frontier node of $N$ and let $m$ be the parent of $n$ in $T$. Let $\overline{z} \subseteq var(N \setminus \{n\})$.*

1. *$var(n) \sqsubseteq_{\mathcal{H}(T,N,\overline{z})} var(m)$.*

*2. $x \notin jv(\mathcal{H}(T,N,\overline{z}))$, for every $x \in (var(n) \setminus var(m))$.*

*Proof.* For reasons of parsimony, let $\mathcal{H} = \mathcal{H}(T,N,\overline{z})$. We first prove (2) and then (1).

(2) Let $x \in var(n) \setminus var(m)$. By Lemma 12(1), $x \notin var(N \setminus \{n\})$. Therefore, $x$ occurs in $var(n)$ in $\mathcal{H}$ and in no other hyperedge. Furthermore, because $\overline{z} \subseteq var(N \setminus \{n\})$, also $x \notin \overline{z}$. Hence $x \notin jv_{\mathcal{H}}(var(n))$.

(1) We need to show that $jv_{\mathcal{H}}(var(n)) \subseteq var(m)$ and $ext_{\mathcal{H}}(var(n) \setminus var(m)) \subseteq var(m)$. Let $x \in jv_{\mathcal{H}}(var(n))$. By contraposition of (2), we know that $x \notin (var(n) \setminus var(m))$. Therefore, $x \in var(m)$ and thus $jv_{\mathcal{H}}(var(n)) \subseteq var(m)$. To show $ext_{\mathcal{H}}(var(n) \setminus var(m)) \subseteq var(m)$, let $y \in ext_{\mathcal{H}}(var(n) \setminus var(m))$. Then $y \notin var(n) \setminus var(m)$ and there exists $\theta \in pred(T,N)$ with $var(\theta) \cap (var(n) \setminus var(m)) \neq \emptyset$ and $y \in var(\theta)$. By Lemma 12(5), $\theta \in pred_T(m \to n)$. Thus, $y \in var(m) \cup var(n)$. Since also $y \notin var(n) \setminus var(m)$, it follows that $y \in var(m)$. Therefore, $ext_{\mathcal{H}}(var(n) \setminus var(m)) \subseteq var(m)$. □

**Lemma 14** *Let $(T,N)$ be a canonical GJT pair and let $n$ be a frontier node of $N$. Then $\mathcal{H}(T,N,\overline{z}) \rightsquigarrow^* \mathcal{H}(T, N \setminus \{n\}, \overline{z})$ for every $\overline{z} \subseteq var(N \setminus \{n\})$.*

*Proof.* For reasons of parsimony, let us abbreviate $\mathcal{H}_1 = \mathcal{H}(T,N,\overline{z})$ and $\mathcal{H}_2 = \mathcal{H}(T, N \setminus \{n\}, \overline{z})$. We make the following case analysis.

Case (1): Node $n$ is the root in $N$. Because the root of a canonical tree is labeled by $\emptyset$ we have $var(n) = \emptyset$. Since $n$ is a frontier node of $N$, $N = \{n\}$. Thus, $hyp(T,N) = \emptyset$ and $hyp(T, N \setminus \{n\}) = \emptyset$. Furthermore, $pred(T,N) = pred(T, N \setminus \{n\}) = \emptyset$ and $\overline{z} \subseteq var(N \setminus \{n\}) = var(\emptyset) = \emptyset$. As such, both $\mathcal{H}_1$ and $\mathcal{H}_2$ are the empty triplet $(\emptyset,\emptyset,\emptyset)$. Therefore $\mathcal{H}_1 \rightsquigarrow^* H_2$.

Case (2): $n$ has parent $m$ in $N$ and $var(m) \neq \emptyset$. Then $var(n) \neq \emptyset$ since in a canonical tree the root node is the only interior node that is labeled by the empty hyperedge. Therefore, $var(n) \in hyp(T,N)$, $var(m) \in hyp(T,N)$, and $var(n) \sqsubseteq_{\mathcal{H}_1} var(m)$ by Lemma 13(1). We can hence apply reduction (CSE) to remove $var(n)$ from $hyp(\mathcal{H}_1)$ and all predicates that intersect with $var(n) \setminus var(m)$ from $pred(\mathcal{H}_1)$. By Lemma 12(2) and 12(6) the result is exactly $\mathcal{H}_2$:

$$
\begin{aligned}
&hyp(\mathcal{H}_2)\\
&= hyp(T, N \setminus \{n\})\\
&= hyp(T,N) \setminus \{var(n)\} = hyp(\mathcal{H}_1) \setminus \{var(n)\}
\end{aligned}
$$

$$
\begin{aligned}
&pred(\mathcal{H}_2)\\
&= pred(T, N \setminus \{n\})\\
&= \{\theta \in pred(T,N) \mid var(\theta) \cap (var(n) \setminus var(m)) = \emptyset\}\\
&= \{\theta \in pred(\mathcal{H}_1) \mid var(\theta) \cap (var(n) \setminus var(m)) = \emptyset\}
\end{aligned}
$$

Case (3): $n$ has parent $m$ in $N$ and $var(m) = \emptyset$. Then $var(n) \neq \emptyset$ since since in a canonical tree the root node is the only interior node that is labeled by the empty hyperedge. By definition of GJTs, it follows that for every $\theta \in pred(m \to n)$ we have $var(\theta) \subseteq var(n) \cup var(m) = var(n)$. In other words: all $\theta \in pred(m \to n)$ are filters. As such, we can use reduction (FLT) to remove all predicates in $pred(m \to n)$ from $\mathcal{H}_1$. This yields a triplet $\mathcal{I}$ with the same hypergraph as $\mathcal{H}_1$, same set of output variables as $\mathcal{H}_1$, and

$$
\begin{aligned}
pred(\mathcal{I}) &= pred(\mathcal{H}_1) \setminus pred_T(m \to n)\\
&= pred(T,N) \setminus pred_T(m \to n)\\
&= pred(T, N \setminus \{n\}) = pred(\mathcal{H}_2),
\end{aligned}
$$

where the third equality is due to Lemma 12(4). We claim that every variable in $e$ is isolated in $\mathcal{I}$. From this the result follows, because then we can apply (ISO) to remove the entire hyperedge $var(e)$ from $hyp(\mathcal{I}) = hyp(\mathcal{H}_1)$ while preserving $out(\mathcal{I})$ and $pred(\mathcal{I})$. The resulting triplet hence equals $\mathcal{H}_2$. To see that $e \subseteq isol(\mathcal{I})$, observe that no predicate in $pred(\mathcal{I}) = pred(T, N \setminus \{n\})$ shares a variable with $var(n) = (var(n) \setminus var(m))$ by Lemma 12(6). Therefore $var(n) \cap var(pred(\mathcal{I})) = \emptyset$. Furthermore, $var(n) \cap jv(\mathcal{I}) = \emptyset$ because $jv(\mathcal{I}) = jv(\mathcal{H}_1)$ and no $x \in var(n) = var(n) \setminus var(m)$ is in $jv(\mathcal{H}_1)$ by Lemma 13(2). □

**Lemma 5** *Let $(T,N_1)$ and $(T,N_2)$ be canonical GJT pairs with $N_2 \subseteq N_1$. Then $\mathcal{H}(T,N_1,\overline{z}) \rightsquigarrow^* \mathcal{H}(T,N_2,\overline{z})$ for every $\overline{z} \subseteq var(N_2)$.*

*Proof.* By induction on $k$, the number of nodes in $N_1 \setminus N_2$. In the base case where $k = 0$, the result trivially holds since then $N_1 = N_2$ and the two triplets are identical. For the induction step, assume that $k > 0$ and the result holds for $k-1$. Because both $N_1$ and $N_2$ are connex subsets of the same tree $T$, there exists a node $n \in N_1$ that is a frontier node in $N_1$, and which is not in $N_2$. Then define $N_1' = N_1 \setminus \{n\}$. Clearly $(T,N_1')$ is again canonical, and $|N_1' \setminus N_2| = k-1$. Therefore, $\mathcal{H}(T,N_1',\overline{z}) \rightsquigarrow^* \mathcal{H}(T,N_2,\overline{z})$ by induction hypothesis. Furthermore, by $\mathcal{H}(T,N_1,\overline{z}) \rightsquigarrow^* \mathcal{H}(T,N_1',\overline{z})$ by Lemma 14, from which the result follows. □

## B.4 Proof of Lemma 6

**Lemma 6** *Let $H_1$ and $H_2$ be two hypergraphs such that for all $e \in H_2$ there exists $\ell \in H_1$ such that $e \subseteq \ell$. Then $(H_1 \cup H_2, \overline{z}, \Theta) \rightsquigarrow^* (H_1, \overline{z}, \Theta)$, for every hyperedge $\overline{z}$ and set of predicates $\Theta$.*

*Proof.* The proof is by induction on $k$, the number of hyperedges in $H_2 \setminus H_1$. In the base case where $k = 0$, the result trivially holds since $H_1 \cup H_2 = H_1$ and the two triplets are hence identical. For the induction step, assume that $k > 0$ and the result holds for $k-1$. Fix some $e \in H_2 \setminus H_1$ and define $H_2' = H_2 \setminus \{e\}$. Then $|H_2' \setminus H_1| = k-1$. We show that $(H_1 \cup H_2, \overline{z}, \Theta) \rightsquigarrow^* (H_1 \cup H_2', \overline{z}, \Theta)$, from which the result follows since $(H_1 \cup H_2', \overline{z}, \Theta) \rightsquigarrow^* (H_1, \overline{z}, \Theta)$ by induction hypothesis. To this end, we observe that there exists $\ell \in H_1 \setminus \{e\}$ with $e \subseteq \ell$. Therefore, $jv_{(H_1 \cup H_2, \overline{z}, \Theta)}(e) \subseteq e \subseteq \ell$. Moreover, $e \setminus \ell = \emptyset$. Therefore, $ext_{(H_1 \cup H_2, \overline{z}, \Theta)}(e \setminus \ell) = \emptyset \subseteq \ell$. Thus $e \sqsubseteq_{(H_1 \cup H_2, \overline{z}, \Theta)} \ell$. We may therefore apply (CSE) to remove $e$ from $H_1 \cup H_2$, yielding $H_1 \cup H_2'$. Since no predicate shares variables with $e \setminus \ell = \emptyset$ this does not modify $\Theta$. Therefore, $(H_1 \cup H_2, \overline{z}, \Theta) \rightsquigarrow^* (H_1 \cup H_2', \overline{z}, \Theta)$. □

# C Proofs of Section 5.2

**Lemma 7** *Let $n$ be a violator of type 1 in $(T,N)$ and assume $(T,N) \xrightarrow{1,n} (T',N')$. Then $(T',N')$ is a GJT pair and it is equivalent to $(T,N)$. Moreover, the number of violators in $(T',N')$ is strictly smaller than the number of violators in $(T,N)$.*

*Proof.* The lemma follows from the following observations. (1) It is straightforward to observe that $T'$ is a valid GJT: the construction has left the set of leaf nodes untouched; took care to ensure that all nodes (including the newly added node $p$) continue to have a guard child; ensures that the connectedness condition continues to hold also for the relocated children of $n$ because every variable in $n$ is present on the entire path between $n$ and $p$; and have ensured that also edge labels remain valid (for the relocated nodes this is because $var(p) = var(g) \subseteq var(n)$).

(2) $N'$ is a connex subset of $T'$ because the subtree of $T$ induced by $N$ equals to subtree of $T'$ induced by $N'$, modulo the replacement of $l$ by $p$ in case that $l$ was in $N$ and $p$ is hence in $N'$.

(3) $(T, N)$ is equivalent to $(T', N')$ because the construction leaves leaf atoms untouched, preserves edge labels, and $var(N) = var(N')$. The latter is clear if $l \notin N$ because then $N = N'$. It follows from the fact that $var(l) = var(p)$ if $l \in N$, in which case $N' = N \setminus \{l\} \cup \{p\}$.

(4) All nodes in $\mathrm{ch}_T(n) \setminus N$ (and their descendants) are relocated to $p$ in $T'$. Therefore, $n$ is no longer a violator in $(T', N')$. Because we do not introduce new violators, the number of violators of $(T', N')$ is strictly smaller than the number of violators of $(T, N)$. □

**Lemma 8** *Let $n$ be a violator of type 2 in $(T, N)$ and assume $(T, N) \xrightarrow{2,n} (T', N')$. Then $(T', N')$ is a GJT pair and it is equivalent to $(T, N)$. Moreover, the number of violators in $(T', N')$ is strictly smaller than the number of violators in $(T, N)$.*

*Proof.* The lemma follows from the following observations. (1) It is straightforward to observe that $T'$ is a valid GJT: the construction has left the set of leaf nodes untouched; took care to ensure that all nodes (including the newly added node $p$) continue to have a guard child; ensures that the connectedness condition continues to hold also for the relocated children of $n$ because every variable in $n$ is also present in $p$, their new parent; and have ensured that also edge labels remain valid (for the relocated nodes this is because $var(p) = var(n)$).

(2) $N'$ is a connex subset of $T'$ because (i) the subtree of $T$ induced by $N$ equals to subtree of $T'$ induced by $N'$ $\{p\}$, (ii) $n \in N$, and (iii) $p$ is a child of $n$ in $T'$. Therefore, $N'$ must be connex.

(3) $(T, N)$ is equivalent to $(T', N')$ because the construction leaves leaf atoms untouched, preserves edge labels, and $var(N) = var(N')$. The latter follows because $var(N') = var(N \cup \{p\})$ and because $var(p) = var(n) \subseteq var(N)$ since $n \in N$.

(4) All nodes in $\mathrm{ch}_T(n) \setminus N$ (and their descendants) are relocated to $p$ in $T'$. Therefore, $n$ is no longer a violator in $(T', N')$. Because we do not introduce new violators, the number of violators of $(T', N')$ is strictly smaller than the number of violators of $(T, N)$. □

## D Description of Competing Systems

**DBToaster.** DBToaster (henceforth denoted DBT) is a state-of-the-art implementation of HIVM. It operates in pull-based mode, and can deal with randomly-ordered update streams. DBT is particularly meticulous in that it materializes only useful views, and therefore it is an interesting implementation for comparison. It has been extensively tested on equijoin queries and has proven to be more efficient than a commercial database management system, a commercial stream processing system and an IVM implementation [30]. DBT compiles given SQL statements into executable trigger programs in different programming languages. We compare against those generated in Scala from the DBToaster Release 2.2[10], and it uses actors[11] to generate events from the input files. During our experiments, however, we have found that this creates unnecessary memory overhead. For a fair memory-wise comparison, we have therefore removed these actors.

[10] https://dbtoaster.github.io/

[11] https://doc.akka.io/docs/akka/2.5/

**Esper.** Esper (E) is a CER engine with a relational model based on Stanford STREAM [4]. It is push-based, and can deal with randomly-ordered update streams. We use the Java-based open source[12] for our comparisons. Esper processes queries expressed in the Esper event processing language (EPL).

**SASE.** SASE (SE) is an automaton-based CER system. It operates in push-based mode, and can deal with temporally-ordered update streams only. We use the publicly available Java-based implementation of SASE[13]. This implementation does not support projections. Furthermore, since SASE requires queries to specify a match semantics (any match, next match, partition contiguity) but does not allow combinations of such semantics, we can only express queries $Q_1$, $Q_2$, and $Q_4$ in SASE. Hence, we compare against SASE for these queries only. To be coherent with our semantics, the corresponding SASE expressions use the any match semantics [3].

**Tesla/T-Rex.** Tesla/T-Rex (T) is also an automaton-based CER system. It operates in push-based mode only, and supports temporally-ordered update streams only. We use the publicly available C-based implementation[14]. This implementation operates in a publish-subscribe model where events are published by clients to the server, known as TRexServer. Clients can subscribe to receive recognized composite events. Tesla cannot deal with queries involving inequalities on multiple attributes e.g. $Q_3$, therefore, we do not show results for $Q_3$. Since Tesla works in a decentralized manner, we measure the update processing time by logging the time at the Tesla TRexServer from the stream start until the end.

**ZStream.** ZStream (Z) is a CER system based on a relational internal architecture. It operates in push-based mode, and can deal with temporally-ordered update streams only. ZStream is not available publicly. Hence, we have created our own implementation following the lazy evaluation algorithm described in the original paper [31]. This paper does not describe how to treat projections, and as such we compare against ZStream only for full join queries $Q_1$–$Q_8$.

[12] http://www.espertech.com/esper/esper-downloads/

[13] https://github.com/haopeng/sase

[14] https://github.com/deib-polimi/TRex