Conjunctive Queries with Inequalities Under Updates

Muhammad Idris
Université Libre de Bruxelles & TU Dresden
midris@ulb.ac.be

Martín Ugarte
Université Libre de Bruxelles
mugartec@ulb.ac.be

Stijn Vansummeren
Université Libre de Bruxelles
svsummer@ulb.ac.be

Hannes Voigt
TU Dresden
hannes.voigt@tu-dresden.de

Wolfgang Lehner
TU Dresden
wolfgang.lehner@tu-dresden.de

ABSTRACT

Modern application domains such as Composite Event Recognition (CER) and real-time Analytics require the ability to dynamically refresh query results under high update rates. Traditional approaches to this problem are based either on the materialization of subresults (to avoid their recomputation) or on the recomputation of subresults (to avoid the space overhead of materialization). Both techniques have recently been shown suboptimal: instead of materializing results and subresults, one can maintain a data structure that supports efficient maintenance under updates and can quickly enumerate the full query output, as well as the changes produced under single updates. Unfortunately, these data structures have been developed only for aggregate-join queries composed of equi-joins, limiting their applicability in domains such as CER where temporal joins are commonplace. In this paper, we present a new approach for dynamically evaluating queries with multi-way θ-joins under updates that is effective in avoiding both materialization and recomputation of results, while supporting a wide range of applications. To do this we generalize Dynamic Yannakakis, an algorithm for dynamically processing acyclic equi-join queries. In tandem, and of independent interest, we generalize the notions of acyclicity and free-connexity to arbitrary θ-joins. We instantiate our framework to the case where θ-joins are only composed of equalities and inequalities ($<, \leq, =, >, \geq$) and experimentally compare this algorithm, called IEDyn, to state of the art CER systems as well as incremental view maintenance engines. IEDyn performs consistently better than the competitor systems with up to two orders of magnitude improvements in both time and memory consumption.

PVLDB Reference Format:
Muhammad Idris, Martin Ugarte, Stijn Vansummeren, Hannes Voigt and Wolfgang Lehner. Conjunctive Queries with Inequalities Under Updates. *PVLDB*, 11(7): 733-745, 2018.
DOI: https://doi.org/10.14778/3192965.3192966

Proceedings of the VLDB Endowment, Vol. 11, No. 7

DOI: https://doi.org/10.14778/3192965.3192966

1. INTRODUCTION

The ability to analyze dynamically changing data is a key requirement of many contemporary applications, usually associated with Big Data, that require such analysis in order to obtain timely insights and implement reactive and proactive measures. Examples of such applications include Financial Systems [12], Industrial Control Systems [19], Stream Processing [34], Composite Event Recognition (CER, also known as Complex Event Processing) [10, 15], and Business Intelligence (BI) [30]. 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 frequent data updates.

In this paper, we focus on the problem of dynamic evaluation for queries that feature multi-way θ-joins in addition to standard equi-joins. To illustrate our setting, consider that we wish to detect potential credit card frauds. Credit card transactions specify their timestamp (ts), the account number (acc), and amount ($amnt$). A typical fraud pattern is that the criminal tests the credit card with a few small purchases to then make larger purchases (cf. [32]). In this respect, we would like to dynamically evaluate the following query, assuming new transactions arrive in a streaming fashion and the pattern must be detected in less than 1 hour.

```
SELECT * FROM Trans as S1, Trans as S2, Trans as L
WHERE S1.ts < S2.ts AND S2.ts < L.ts AND L.ts < S1.ts + 1h
AND S1.acc = S2.acc AND S2.acc = L.acc
AND S1.amnt < 100 AND S2.amnt < 100 AND L.amnt > 400
```

Queries like this with inequality joins appear in both CER and BI scenarios. Traditional techniques to process these queries dynamically can be categorized in two approaches: relational and automaton-based. We next discuss both approaches and their drawbacks.

Relational. Relational approaches such as [3, 25, 26] are based on a form of *Incremental View Maintenance* (IVM). To process a query Q over a database db, IVM techniques materialize the output $Q(db)$ and evaluate *delta queries*. Upon update u, delta queries use db, u and the materialized $Q(db)$ to compute the set of tuples to add/delete from $Q(db)$ in order to obtain $Q(db+u)$. If u is small w.r.t. db, this is expected to be faster than recomputing $Q(db+u)$ from scratch. To further speed up dynamic query processing, we may materialize not only $Q(db)$ but also the result of some subqueries. This is known as Higher-Order IVM (HIVM) [24, 25, 27]. (H)IVM present important drawbacks,

however. First, materialization of $Q(db)$ requires $\Omega(\|Q(db)\|)$ space, where $\|db\|$ denotes the size of db. Therefore, when $Q(db)$ is large compared to db, materializing $Q(db)$ quickly becomes impractical, especially for main-memory based systems. HIVM is even more affected by this problem because it not only materializes the result of Q but also the results to some subqueries. For example, in our fraud query HIVM would materialize the results of the following join in order to respond quickly to the arrival of a potential transaction L:

$$\sigma_{\text{amnt}<100}(S_1) \bowtie_{S1.ts<S2.ts \wedge S1.acc=S2.acc} \sigma_{\text{amnt}<100}(S_2) \quad (\star)$$

If we assume that there are N small transactions in the time window, all of the same account, this materialization will take $\Theta(N^2)$ space. This becomes rapidly impractical when N becomes large, specially in a main-memory based setting.

Automata. Automaton-based approaches (e.g., [2, 9, 13, 14, 36, 39]) are primarily employed in CER systems. In contrast to the relational approaches, they assume that the arrival order of event tuples corresponds to the timestamp order (i.e., there are no out-of-order events) and build an automaton to recognize the desired temporal patterns in the input stream. There are two automata-based recognition approaches. In the first approach, followed by [2, 36], 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 the temporal constraints need no longer be checked during the search, the additional constraints (in our example, $L.ts < S1.ts + 1h$ and $S_1.\text{acc} = S_2.\text{acc} = L.\text{acc}$) must still be verified. If the additional constraints are highly selective this approach creates an unnecessarily large update latency, given that each event triggering a transition to a final state may cause re-evaluation of a sub-join on the cached data, only to find that few tuples contribute to the output. In the second approach, followed by [9, 13, 14, 39], partial runs are materialized according to the automaton's topology. For our example query, this means that, just like HIVM, the join $(\star)$ is materialized and maintained so it is available when a large amount transaction L arrives. This approach hence shares with HIVM its high memory overhead and maintenance cost.

It has been recently shown that the drawbacks of these two approaches can be overcome by a rather simple idea [22, 28]. Instead of materializing results and subresults, one can keep a compressed representation of the output that supports efficient maintenance under updates. This representation can generate the output spending only a constant amount of work to produce each new result tuple, which makes it competitive with enumeration from a fully materialized output. This idea was first presented by a subset of the authors [22] in the Dynamic Yannakakis Algorithm (Dyn), an algorithm for efficiently processing acyclic aggregate-join queries under updates that is worst-case optimal for two classes of queries (q-hierarchical and free-connex acyclic conjunctive queries). A different approach named F-IVM, based on so-called *factorized databases*, was later developed to dynamically process aggregate-join queries that are not necessarily acyclic [28].

Unfortunately, both Dyn and F-IVM are only applicable to queries with equality joins, and as such they do not support analytical queries with other types of joins like the ones with inequalities ($\leq, <, \geq, >$). Therefore, the current state of the art techniques for dynamically processing queries with joins beyond equality suffer either from a high update latency (if subresults are not materialized) or a high memory footprint (if subresults are materialized).

In this paper, we overcome these problems by generalizing the Dynamic Yannakakis Algorithm to conjunctive queries with arbitrary θ-joins. We show that, in the specific case of inequality joins, this generalization improves the state of the art for dynamically processing inequality joins by performing consistently better, with up to two orders of magnitude improvements in processing time and memory consumption.

Contributions. We focus on the class of Generalized Conjunctive Queries (GCQs for short), i.e., conjunctive queries with θ-joins, evaluated under multiset semantics.

(1) We devise a succinct and efficiently updatable data structure to dynamically process GCQs. To this end, we first generalize the notions of acyclicity and free-connexity to queries with arbitrary θ-joins (Section 3). Our data structure degrades gracefully: if a GCQ only contains equalities our approach inherits the worst-case optimality provided by Dyn.

(2) We present GDyn, a general framework for extending Dyn to free-connex acyclic GCQs. Our treatment is general in the sense that the θ-join predicates are treated abstractly. GDyn hence applies to all predicates. We analyze the complexity of GDyn, and identify properties of indexing structures that are required in order for GDyn to support constant delay enumeration of results as well as efficient update processing (Section 5).

(3) We instantiate GDyn to the particular case of inequality and equality joins. We show that updates can be processed in log-linear time and results can be enumerated with logarithmic delay. Moreover, if there is at most one inequality between any pair of relations, then results can be enumerated with constant delay. We call the resulting algorithm IEDyn. We first illustrate this algorithm by means of an extensive example (Section 4), and then describe the required data structures formally at the end of Section 5.

(4) We experimentally compare IEDyn against state of the art HIVM and CER frameworks. Our extensive experiments show that IEDyn performs consistently better, with up to two orders of magnitude improvements in both speed and memory consumption (Section 6 and Section 7).

We introduce the required background in Section 2 and discuss future work in Section 6. Proofs of formal statements are omitted because of space constraints.

Additional related work. In addition to the work already cited on CER and (H)IVM, our setting is closely related to query evaluation with constant delay enumeration [4–6, 8, 22, 28, 29, 29, 31, 33]. This setting, however, deals with equi-joins only. Also related, although restricted to the static setting, is the practical evaluation of binary [16, 17, 20] and multi-way [7, 38] inequality joins. Our work, in contrast, considers dynamic processing of multi-way θ-joins, with a specialization to inequality joins. Recently, Khayyat et al. [23] 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 in the case of multi-way joins. As a result, they either materialize subresults (implying a space overhead that can be more than linear), or recompute subresults. We do neither.

2. PRELIMINARIES

Traditional conjunctive queries are cross products between relations, restricted by equalities. Generalized conjunctive queries (GCQs) are cross products between relations, but restricted by arbitrary predicates. We use the following notation for queries.

Query Language. Throughout the paper, let $\overline{x}$, $\overline{y}$, ..., denote finite sets of *variables* (which will take the role of *attributes*), and let $x, y, z, \ldots$ denote individual variables. A GCQ is an expression of the form

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

Here $r_1, \ldots, r_n$ are *relation symbols*; $\overline{x_1}, \ldots, \overline{x_n}$ are finite sets of variables (of the same arity as $r_1, \ldots, r_n$); $P_1, \ldots, P_m$ 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 P of tuples over $\overline{x}$. For example, $P(x, y) = x < y$ is the set of all tuples (a, b) satisfying $a < b$. For convenience, we abbreviate $\vec{t} \in P$ by $P(\vec{t})$ and indicate that P is a predicate over $\overline{x}$ by writing $P(\overline{x})$.

Example 2.1. The following query is hence a GCQ.

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

Intuitively, the query specifies that we should take the natural join[1] of $r(x, y)$, $s(y, z, w)$, and $t(u, v)$, and from this result filter those tuples that satisfy $x < z$ and $w < u$.

We call $\overline{y}$ the *output variables* of GCQ 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. The elements $r_i(\overline{x_i})$ are called *atomic queries* (or *atoms*). We write $at(Q)$ for the set of all atoms in Q, and $pred(Q)$ for the set of all predicate in Q. A *normal conjunctive query* (NCQ for short) is a GCQ where $pred(Q) = \emptyset$.

Semantics. We evaluate GCQs over Generalized Multiset Relations (GMRs for short) [22, 24, 25]. A GMR over $\overline{x}$ is a relation R over $\overline{x}$ (i.e., a finite set of tuples with schema $\overline{x}$) in which each tuple $\vec{t}$ is associated with a non-zero integer multiplicity $R(\vec{t}) \in \mathbb{Z} \setminus \{0\}$.[2] 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 $\text{supp}(R)$ for the finite set of all tuples in R; $\vec{t} \in R$ to indicate $\vec{t} \in \text{supp}(R)$; and $|R|$ for $|\text{supp}(R)|$. A GMR R is *positive* if $R(\vec{t}) > 0$ for all $\vec{t} \in \text{supp}(R)$.

The operations of GMR union $(R+S)$, minus $(R-S)$, projection $(\pi_{\overline{z}} R)$, natural join $(R \bowtie T)$[1] and selection $(\sigma_P(R))$ are defined similarly as in relational algebra with multiset semantics. Figure 2 illustrates these operations. We refer to [22, 25] for a formal semantics. We abbreviate $\sigma_P(R \bowtie T)$ by $R \bowtie_P T$, and if $\overline{x}$ is the set of attributes of R we abbreviate $\pi_{\overline{x}}(R \bowtie_P T)$ by $R \ltimes_P T$.

[1]As usual, the natural join between relations that have a disjoint schema is simply their cartesian product.

[2]In their full generality, GMRs can carry multiplicities that are taken from an arbitrary algebraic ring structure (cf., [24]), which can be useful to describe the computation of aggregations over the result of a GCQ. To keep the notation and discussion simple, we fix the ring $\mathbb{Z}$ of integers throughout the paper but our result generalize trivially to arbitrary rings.

R

x	y	z	$\mathbb{Z}$
1	2	2	2
2	4	6	3
1	2	3	3

S

u	v	$\mathbb{Z}$
4	5	5
2	3	4
1	4	2

T

u	v	$\mathbb{Z}$
4	5	−4
2	1	6
1	4	3

$S \bowtie T$

u	v	$\mathbb{Z}$
4	5	−20
1	4	6

$\pi_y(R)$

y	$\mathbb{Z}$
2	5
4	3

$S + T$

u	v	$\mathbb{Z}$
4	5	1
2	3	4
1	4	5
2	1	6

$S - T$

u	v	$\mathbb{Z}$
4	5	9
2	3	4
1	4	−1
2	1	−6

$R \bowtie_{y<u} S$

x	y	z	u	v	$\mathbb{Z}$
1	2	2	4	5	10
1	2	3	4	5	15

Figure 1: Operations on GMRs

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: construct the natural join of all GMRs in the database, do a selection over the result w.r.t. each predicate, and finally project on $\overline{y}$.

Updates and deltas. An *update* to a GMR R is simply a GMR ΔR over the same variables as R. Applying update ΔR 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. The positiveness condition ensures that after applying u to db we obtain again a valid database. We write $db + u$ for the database obtained by applying u to each atom of db, i.e., $(db + u)_{r(\overline{x})} = db_{r(\overline{x})} + u_{r(\overline{x})}$, for every atom $r(\overline{x})$ of db. For every query Q, every database db and every update u to db, we define the delta query $\Delta Q(db, u)$ of Q w.r.t. db and u by $\Delta Q(db, u) := Q(db + u) - Q(db)$. As such, $\Delta Q(db, u)$ is the update that we need to apply to $Q(db)$ in order to obtain $Q(db + u)$.

Enumeration with bounded delay. 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 output; the time between any two consecutive outputs; and the time between the last output 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 $E_R = \{(\vec{t}, R(\vec{t})) \mid \vec{t} \in \text{supp}(R)\}$. When evaluating a GCQ Q, we will be interested in representing the outputs of Q by means of a family $\mathcal{D}$ of data structures, one data structure $D_{db} \in \mathcal{D}$ for each input database db. We say that Q *can be enumerated from* $\mathcal{D}$ *with delay* f, if for every input db we can enumerate $Q(db)$ from D_{db} with delay $O(f(D_{db}))$, where f assigns a natural number to each D_{db}. Intuitively f measures D_{db} in some way. In particular, if f is constant we say the results are generated from the data structure with constant-delay enumeration (CDE).

As a trivial example of CDE of a GCQ Q, assume that, for each db, the pairs $(\vec{t}, Q(db)(\vec{t}))$ of $Q(db)$ are stored in an array A_{db} (without duplicates). Then the family (A_{db}) supports CDE of Q: ENUM(A_{db}) simply iterates over each element in A_{db}, one by one, always outputting the current element. Since array indexation is a $O(1)$ operation, this hence gives constant delay. CDE of a GCQ Q can hence always be done naively by materializing $Q(db)$ in an in-memory array A_{db}. Unfortunately, A_{db} requires memory proportional to $\|Q(db)\|$

which, depending on Q, can be of size polynomial in $\|db\|$. We hence search for other data structures that can represent $Q(db)$ using less space, while still allowing enumeration with the same (worst-case) complexity as enumeration from a materialized array: namely, with constant delay.

Computational Model. It is important to note that we focus on dynamic query evaluation in main memory and measure time and space under data complexity [35]. That is, the query is considered to be fixed and not part of the input. This makes sense under dynamic query evaluation, where the query is known in advance and the data is constantly changing. In particular, the number of relations to be queried, their arity, and the length of the query are all constant.

Furthermore, we assume a computational model where the space used by tuple values and integers, the time of arithmetic operations on integers, and the time of memory lookups are all $O(1)$. We further assume that every GMR R can be represented by a data structure that allows (1) enumeration of R with constant delay; (2) multiplicity lookups $R(\vec{t})$ in $O(1)$ time given $\vec{t}$; (3) single-tuple insertions and deletions in $O(1)$ time; while (4) having a size that is proportional to the number of tuples in the support of R. Essentially, our assumptions amount to perfect hashing of linear size [11]. Although this is not realistic for practical computers, it is well known that complexity results for this model can be translated, through amortized analysis, to average complexity in real-life implementations [11].

3. GENERALIZED ACYCLICITY

Join queries are GCQs without projections that feature equality joins only. The well-known subclass of acyclic join queries [1, 37], in contrast to the entire class of join queries, can be evaluated in time $O(\|db\| + \|Q(db)\|)$, i.e., linear in both input and output. This result relies on the fact that acyclic join queries admit a tree structure that can be exploited during evaluation. In previous work [22], we showed that this tree structure can also be exploited for efficient processing of NCQs under updates. In this section, we therefore extend the tree structure and the notion of acyclicity from join queries to GCQs with both projections and arbitrary θ-joins. We begin by defining this tree structure and the related notion of acyclicity for full GCQs. Then, we proceed with the notion corresponding to GCQs that feature projections, known as free-connex acyclicity. Finally, we discuss how to compute these tree structures for a given GCQ Q.

Generalized Join Trees. To simplify notation, we denote the set of all variables (resp. atoms, resp. predicates) that occur in a mathematical 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 $var(n)$ denotes the set of variables occurring in the label of n, and similarly for edges and trees themselves.

Definition 3.1 (GJT)**.** Let $\mathcal{A}$ be a finite set of atoms and let $\mathcal{P}$ be a finite set of predicates. A *hyperedge* in $\mathcal{A}$ is a set $\overline{x}$ of variables such that there exists $r(\overline{y}) \in \mathcal{A}$ with $\overline{x} \subseteq \overline{y}$. A *Generalized Join Tree* for $\mathcal{A}$ and $\mathcal{P}$ is a node-labeled and edge-labeled directed tree $T = (V, E)$ such that:

- Every leaf is labeled by an atom in $\mathcal{A}$ and every atom in $\mathcal{A}$ occurs as the label of exactly one leaf in T.
- Every interior node n is labeled by a hyperedge in $\mathcal{A}$ and has at least one child c s.t. $var(n) \subseteq var(c)$.

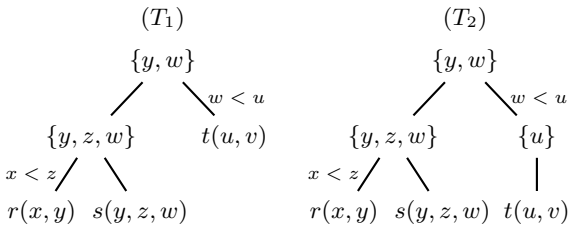

Figure 2: Two example GJTs.

- 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 undirected path linking m and n.
- Every edge $p \to c$ from parent p to child c in T is labeled by a set $pred(p \to c) \subseteq \mathcal{P}$ of predicates. It is required that for every predicate $P(\overline{z}) \in pred(p \to c)$ we have $\overline{z} \subseteq var(p) \cup var(c)$. Moreover, every predicate in $\mathcal{P}$ must occur in the label of at least one edge of T.

Figure 2 illustrates two GJTs for $\mathcal{P} = \{x < z, w < u\}$ and $\mathcal{A} = \{r(x, y), s(y, z, w), t(u, v)\}$.

Definition 3.2. A GCQ Q is *acyclic* if there is a GJT for $at(Q)$ and $pred(Q)$. It is *cyclic* otherwise.

Example 3.3. The join trees in Figure 2 hence show that the following full GCQ is acyclic.

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

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

If Q does not contain any predicates, that is, if Q is a NCQ, then the last condition of Definition 3.1 vacuously holds. In that case, the definition corresponds to the definition of a generalized join tree given in [22], where it was also shown that a NCQ is acyclic under any of the traditional definitions of acyclicity (e.g., [1]) if and only if the query has a GJT for $at(Q)$. In this sense, Definition 3.2 indeed generalizes acyclicity from NCQs to GCQs.

Free-connex acyclicity. Acyclicity is actually a notion for full GCQs. Indeed, note that whether or not Q is acyclic does not depend on the projections of Q (if any). To also process queries with projections efficiently, a related structural constraint known as free-connex acyclicity is required.

Definition 3.4 (Connex, Frontier)**.** Let $T = (V, E)$ be a GJT. 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. 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.

To illustrate, $\{\{y, w\}, \{u\}, \{y, z, w\}\}$ is a connex subset of T_2 of Figure 2. Its frontier is $\{\{y, z, w\}, \{u\}\}$. In contrast, $\{\{y, w\}, \{y, z, w\}, t(u, v)\}$ is not a connex subset of T_2.

Definition 3.5. (Compatible, Free-Connex Acyclic) Let T be a GJT. A GCQ Q is *compatible with* T if T is a GJT for Q and T has a connex subset N with $var(N) = out(Q)$. A GCQ is *free-connex acyclic* if it has a compatible join tree.

In particular, every full acyclic GCQ is free-connex acyclic since the entire set of nodes V of a GJT T for Q is a connex set with $var(V) = out(Q)$.

Example 3.6. Let $Q_2 = \pi_{y,z,w,u}(Q_1)$ with Q_1 the GCQ from Example 3.3. Then Q_2 is free-connex acyclic since it is compatible with GJT T_2 from Figure 2. By contrast, Q_2 is not compatible with GJT T_1 since any connex set of T_1 that includes a node with variable u will also include variable v, which is not in $out(Q_2)$. Finally, it can be verified that no GJT for Q_1 is compatible with $\pi_{x,u}(Q_1)$; the latter query is hence not free-connex acyclic.

We can efficiently check (free-connex) acyclicity.

Proposition 3.7. *There is a polynomial time algorithm that, given a GCQ Q, returns a GJT for Q if Q is acyclic and* `Null` *otherwise. Moreover, if Q is free-connex acyclic the returned tree T is compatible with Q.*

For the sake of space we omit this algorithm.

Semantic acyclicity. It is important to observe that acyclicity as defined above is a *syntactic* notion. That is, it is possible that a GCQ Q itself is not acyclic but that it is equivalent to a GCQ Q' that is acyclic. For example, consider

$$Q_3 = \big(r(x,y) \bowtie s(y,z,w) \bowtie t(u,v) \mid x < z \wedge z < u \wedge x < u\big)$$

It can be verified that Q_3 is cyclic. Note, however, that Q_3 is equivalently expressed by the acyclic GCQ

$$Q_4 = \big(r(x,y) \bowtie s(y,z,w) \bowtie t(u,v) \mid x < z \wedge z < u\big).$$

The problem arises here because Q_3 contains a predicate that is redundant: $x < u$ is implied by $x < z$ and $z < u$, and therefore it could be eliminated. We say that a GCQ Q is *semantically acyclic* if it is equivalent to an acylic GCQ Q'. As such, Q_3 is semantically acyclic. Semantic free-connex acyclicity is defined similarly.

The fact that acyclicity is a syntactic notion is problematic from a systems perspective, since we may not be able dynamically process GCQ Q directly: the techniques of Sections 4 and 5 require a compatible GJT for Q (which hence needs to be acyclic). If Q is semantically but not syntactically acyclic, we would hence like to transform Q into an equivalent acyclic query Q' and process Q' instead. For the specific setting where Q uses only equality ($=$) and inequality predicates ($\leq, <, \geq, >$), we have the following.

Proposition 3.8. *There is a polynomial time algorithm that, given a semantically acyclic GCQ Q, returns an acyclic GCQ Q' such that $Q \equiv Q'$. Moreover, if Q semantically free-connex acyclic then Q' is free-connex acyclic.*

Binary and Strongly Compatible GJTs. In order to simplify the presentation of what follows, we will focus on the class of binary and strongly compatible GJTs in the rest of the paper. A *binary GJT* is a GJT in which every node has at most two children. A GJT T is *strongly compatible* with a GCQ Q if it is a GJT for Q and has a sibling-closed connex subset N such that $var(N) = out(Q)$. Here, N is called *sibling-closed* if for every node $n \in N$ with a sibling m, m is also in N. It can be shown that if N is sibling-closed, then $var(N) = var(F)$ where F is the frontier of N. The following proposition shows that we can always convert to binary and strongly compatible GJTs. Therefore, there is no loss of generality by restricting to these classes.

Proposition 3.9. *There is a polynomial time algorithm that, given a GJT T' for a query Q, returns a binary GJT T' for Q. Moreover, if T is compatible with Q, then T' is strongly compatible with Q.*

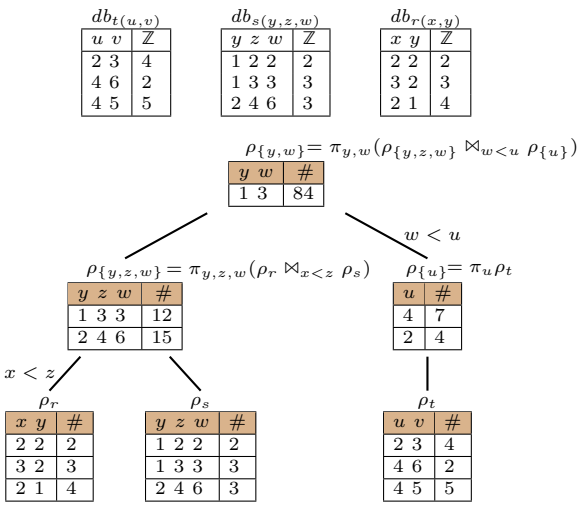

Figure 3: Example database and its T_2-reduct.

4. DYNAMIC JOINS WITH EQUALITIES AND INEQUALITIES: AN EXAMPLE

In this section we illustrate how to dynamically process free-connex acyclic GCQs when all predicates are inequalities ($\leq, <, \geq, >$). We do so by means of an extensive example that shows the required indexing structures and GMRs. The full definitions and algorithms (that apply to arbitrary θ-joins) will be formally presented in Section 5.

Throughout this section we consider the following query, which was shown free-connex acyclic in Example 3.6:

$$Q = \pi_{y,z,w,u}\big(r(x,y) \bowtie s(y,z,w) \bowtie t(u,v) \mid x < z \wedge w < u\big).$$

Let T_2 be the GJT from Figure 2. We will process Q based on a *T_2-reduct*, a data structure that succinctly represents the output of Q evaluated over the current database. For every node n, define $pred(n)$ as the set of all predicates on child edges of n, i.e. $pred(n) = \bigcup_{c \text{ child of } n} pred(n \to c)$.

Definition 4.1 (T-reduct)**.** Let T be a GJT for a query Q and let db be a database over $at(Q)$. The T-reduct (or *semi-join reduction*) of db is a collection ρ of GMRs, one GMR ρ_n for each node $n \in T$, defined inductively as follows:
- if $n = r(x)$ is an atom, then $\rho_n = db_{r(x)}$
- if n has a single child c, then $\rho_n = \pi_{var(n)}\,(\rho_c \mid pred(n))$
- otherwise, n has two children c_1 and c_2. In this case we have $\rho_n = \pi_{var(n)}\,(\rho_{c_1} \bowtie \rho_{c_2} \mid pred(n))$.

Figure 3 depicts an example database (top) and its T_2-reduct ρ (bottom). Note, for example, that the only tuple in the GMR at the root ($\rho_{\{y,w\}}$) comes from the join of $\rho_{\{y,z,w\}}$ and $\rho_{\{u\}}$ restricted to $w < y$ and projected over $\{y, w\}$.

It is important to observe that the size of a T-reduct of a database db can be at most linear in the size of db. To see why, observe that, as illustrated in Figure 3, for each node n there is some descendant atom α (possibly n itself) such that $\text{supp}(\rho_n) \subseteq \text{supp}(\pi_{var(n)}\, db_\alpha)$. Note that $Q(db)$, in contrast, can become cubic in the size of db in the worst case.

Enumeration. From a T-reduct we can enumerate the result $Q(db)$ rather naively simply by recomputing the query results, in particular because we have access to the complete database in the leaves of T. We would like, however, to make

the enumeration as efficient as possible. To this end, we equip T-reducts with a set of indices. To avoid the space cost of materialization, we do not want the indices to use more space than the T-reduct itself (i.e., linear in db). We illustrate these ideas in our running example by introducing a simple set of indices that allow for constant-delay enumeration.

Let $N = \{\{y, w\}, \{y, z, w\}, \{u\}\}$ be the sibling-closed subset of T_2 with $var(N) = out(Q) = \{y, z, w, u\}$. We rely on the strong compatibility of T_2 with Q without loss of generality (see Proposition 3.9). To enumerate the query results, we will traverse top-down the nodes in N. The traversal works as follows: for each tuple t_1 in $\rho_{\{y,w\}}$, we consider all tuples t_2 in $\rho_{\{y,z,w\}}$ that are compatible with t_1, and all tuples $t_3 \in \rho_{\{u\}}$ that are compatible with t_1. Compatibility here means that the corresponding equalities and inequalities are satisfied. Then, for each pair (t_2, t_3), we output the tuple $t_2 \cup t_3$ with multiplicity $\rho_{\{y,z,w\}}(t_2) \times \rho_{\{u\}}(t_3)$. A crucial difference here with naive recomputation is that, since $\rho_{\{y,w\}}$ is already a join between $\rho_{\{y,z,w\}}$ and $\rho_{\{u\}}$, we will only iterate over *relevant* tuples: each tuple that we iterate over will produce a new output tuple. For example, we will never look at the tuple $\langle y : 2, z : 4, w : 6\rangle$ in $\rho_{\{y,z,w\}}$ because it does not have a compatible tuple at the root.

To implement this enumeration strategy with constant delay, we desire index structures on $\rho_{\{y,z,w\}}$ and $\rho_{\{u\}}$ that allow to enumerate, for a given tuple t_1 in $\rho_{\{y,w\}}$, all compatible tuples $t_2 \in \rho_{\{y,z,w\}}$ (resp. $t_3 \in \rho_{\{u\}}$) with constant delay. In the case of $\rho_{\{u\}}$ this is achieved simply by keeping $\rho_{\{u\}}$ sorted decreasingly on variable u. Given tuple t_1, we can enumerate the compatible tuples from $\rho_{\{u\}}$ by iterating over its tuples one by one in a decreasing manner, starting from the largest value of u, and stopping whenever the current u value is smaller or equal than the w value in t_1. For indexing $\rho_{\{y,z,w\}}$ we require a more standard index. In particular, we need to enumerate all tuples that have the same y and w value as t_1. CDE can be easily achieved by using a hash-based index on y and w. This index is depicted as $I_{\rho_{\{y,z,w\}}}$ in Figure 4. It is not hard to see that, since the described indices provide CDE of the compatible tuples given t_1, the described strategy provides CDE of $Q(db)$.

Updates. Next we illustrate how to process updates. The objective here is to transform the T_2-reduct of db into a T_2-reduct of $db+u$, where u is the received update. To do this efficiently we use additional indexes on ρ. We present the intuitions behind these indices with an update consisting of two insertions: $\langle y : 2, z : 3, w : 6\rangle$ with multiplicity 2 and $\langle u : 4, v : 9\rangle$ with multiplicity 3. Figure 4 depicts the update process, highlighting in gray the update modifications.

Let us first discuss how the tuple $t_1 = \langle y : 2, z : 3, w : 6\rangle$ is processed. We proceed bottom-up, starting at ρ_s which is itself affected by the insertion of t_1. Subsequently, we need to *propagate* the modification of ρ_s to its ancestors $\rho_{\{y,z,w\}}$ and $\rho_{\{y,w\}}$. Concretely, from the definition of T-reduction, it follows that we need to add the following modifications to ρ_s, $\rho_{\{y,z,w\}}$, and $\rho_{\{y,w\}}$ on t_1:

$\Delta\rho_s = [t_1 \mapsto 2]$,
$\Delta\rho_{\{y,z,w\}} = \pi_{y,z,w}\,(\rho_r \bowtie_{x<z} \Delta\rho_s)$,
$\Delta\rho_{\{y,w\}} = \pi_{y,w}\left(\Delta\rho_{\{y,z,w\}} \bowtie_{w<u} \rho_{\{u\}}\right)$.

To compute the joins on the right-hand sides efficiently, we create a number of additional indexes on ρ_r, ρ_s, and $\rho_{\{y,z,w\}}$. Concretely, in order to compute $\pi_{y,z,w}\,(\rho_r \bowtie_{x<z} \Delta\rho_s)$ efficiently we *group* tuples in the GMR ρ_r by the variables that ρ_r has in common with ρ_s (in this case y) and then,

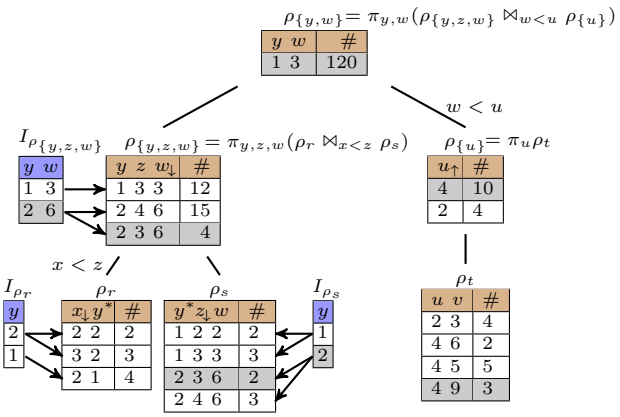

Figure 4: T_2-rep of $db+u$. u is the update containing the tuples $(\langle y : 2, z : 3, w : 6\rangle, 2)$ and $(\langle u : 4, v : 9\rangle, 3)$. T_2 and db are as in Figure 3.

per group, *sort* tuples ascending on variable x. We mark grouping variables in Figure 4 with $*$ (e.g. y^*), and sorting by $\downarrow$ (for ascending, e.g., $x_\downarrow$) and $\uparrow$ (for descending). A hash index on the grouping variables (denoted I_{ρ_r} in Figure 4) then allows to find the group given a y value. The join can then be processed by means of a hybrid form of sort-merge and index nested loop join. Sort $\Delta\rho_s$ ascendingly on y and z. For each y-group in $\Delta\rho_s$ find the corresponding group in ρ_r by passing the y value to the index I_{ρ_r}. Let t' be the first tuple in the $\Delta\rho_s$ group. Then iterate over the tuples of the ρ_r group in the given order and sum up their multiplicities until x becomes larger than $t'(z)$. Add t' to the result with its original multiplicity multiplied by the found sum (provided it is non-zero). Then consider the next tuple in the $\Delta\rho_s$ group, and continue summing from the current tuple in the ρ_r group until x becomes again larger than z, and add the result tuple with the correct multiplicity. Continue repeating this process for each tuple in the $\Delta\rho_s$ group, and for each group in $\Delta\rho_s$. In our case, there is only one group in $\Delta\rho_s$ (given by $y = 2$) and we will only iterate over the tuple $\langle x : 2, y : 2\rangle$ in ρ_r, obtaining a total multiplicity of 2, and therefore compute $\Delta\rho_{\{y,z,w\}} = [t_1 \to 4]$. In order to compute the join $\pi_{y,w}\left(\Delta\rho_{\{y,z,w\}} \bowtie_{w<u} \rho_{\{u\}}\right)$ efficiently, we proceed similarly. Here, however, there are no grouping variables on $\rho_{\{u\}}$ and it hence suffices to sort $\rho_{\{u\}}$ descendingly on u. Note that this was actually already required for efficient enumeration. Also note that $\Delta\rho_{\{y,w\}}$ is in fact empty.

Now we discuss how to process $t_2 = \langle u : 4, v : 9\rangle$. First, we insert t_2 into ρ_t. We need to propagate this change to the parent $\rho_{\{u\}}$ by calculating $\Delta\rho_{\{u\}} = \pi_u\,\Delta\rho_t$. This can be done by a simple hash-based aggregation. Finally, we need to propagate $\Delta\rho_{\{u\}}$ to the root by computing the delta $\Delta\rho_{\{y,w\}} = \pi_{y,w}(\rho_{\{y,z,w\}} \bowtie_{w<u} \Delta\rho_{\{u\}})$. To process this join efficiently we proceed as before. Again, there are no grouping variables on $\rho_{\{y,z,w\}}$ (since it has no variables in common with $\rho_{\{u\}}$) and it hence suffices to sort $\rho_{\{y,z,w,\}}$ ascending on w. The only tuple that we iterate over during the hybrid join is $\langle y : 1, z : 3, w : 3\rangle$ wich has multiplicity 12. Hence, we have $\Delta\rho_{y,w} = [\langle y : 1, w : 3\rangle \mapsto 36\ ]$, concluding the example.

In this section we have presented an extensive example for dynamically processing a query involving equalities and inequalities. In the next section, we first present a formal

and general framework to process free-connex acyclic GCQs and then instantiate it to the case of equalities and inequalities. Having a formal framework we study the complexity of update processing and enumeration of results.

5. DYNAMIC YANNAKAKIS OVER GCQS

Dynamic Yannakakis (Dyn) is an algorithm to efficiently evaluate free-connex acyclic aggregate-equijoin queries under updates [22]. This algorithm matches two important theoretical lower bounds (for q-hierarchical NCQs [6] and free-connex acyclic NCQs [4]), and is highly efficient in practice. In this section we present a generalization of Dyn, called GDyn, to dynamically process free-connex acyclic GCQs. Since predicates in a GCQ can be arbitrary, our approach is purely algorithmic; how efficiently can GDyn process updates and produce results will depend entirely on the efficiency of the underlying data structures. Here we only describe the properties that those data structures should satisfy and present the general (worst-case) complexity of the algorithm. The techniques and indices presented in the previous section provide a practical instantiation of GDyn to the case of equalities and inequalities; in this section we make a parallel between that instantiation and the more abstract definitions of GDyn.

Throughout this section let Q be a free-connex acyclic GCQ, let T be a strongly-compatible binary GJT for Q. Like in the case of equalities and inequalities, the dynamic processing of Q will be mainly based on a T-reduct of the current database db. A set of indices will then be added to optimize the enumeration of query results and maintenance of the T-reduct under updates. We formalize the notion of index as follows:

Definition 5.1 (Index)**.** Let R be a GMR over $\overline{x}$, let $\overline{y}$ be a set of variables, let $\overline{w}$ be a set of variables satisfying $\overline{w} \subseteq \overline{x} \cup \overline{y}$, and let $P(\overline{z})$ be a predicate with $\overline{z} \subseteq \overline{x} \cup \overline{y}$. An *index* on R by $(P, \overline{y}, \overline{w})$ with delay f is a data structure I that provides, for any given GMR $R_{\overline{y}}$ over $\overline{y}$, enumeration of $\pi_{\overline{w}}(R \bowtie R_{\overline{y}} \mid P)$ with delay $O(f(|R| + |R_{\overline{y}}|))$. The update time of index I is the time required to update I to an index on $R + \Delta R$ (by $(P, \overline{y}, \overline{w})$) given update ΔR to R.

For example, I_{ρ_r} in Figure 4 is used as an index on ρ_r by $(x < z, \{y, z, w\}, \{y, z, w\})$. Indeed, in the previous section we precisely discussed how I_{ρ_r} allows to efficiently compute $\pi_{y,z,w}(\rho_r \bowtie_{x<z} \Delta\rho_s)$ for an update $\Delta\rho_s$ to ρ_s. Having defined the notion of index, we proceed to discuss how GDyn enumerates query results and processes updates.

Enumeration. Let db be the current state of the database and let N be the sibling-closed connex subset of T such that $var(N) = out(Q)$. Such N exists since T is strongly compatible with Q. To enumerate $Q(db)$ from a T-reduct ρ of db we can iterate over the reductions ρ_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 ρ_n. For every such tuple $\vec{t}$, we iterate only over the tuples in the children c of n that are compatible with $\vec{t}$ (i.e., tuples in ρ_c that join with $\vec{t}$ and satisfy $pred(n \to c)$). This procedure continues until we reach nodes in the frontier of N at which time the output tuple can be constructed. The pseudocode for enumeration is given in Algorithm 1. There, the tuples that are compatible with $\vec{t}$ are computed by $\rho_c \ltimes_{pred(n\to c)} \vec{t}$.

Algorithm 1 Enumerate $Q(db)$ given T-reduct ρ of db.

```
1: function ENUM_{T,N}(ρ)
2:   for each t⃗ ∈ ρ_{root(T)} do ENUM_{T,N}(root(T), t⃗, ρ)

3: function ENUM_{T,N}(n, t⃗, ρ)
4:   if n is in the frontier of N then yield (t⃗, ρ_n(t⃗))
5:   else if n has one child c then
6:     for each s⃗ ∈ ρ_c ⋉_{pred(n→c)} t⃗ do ENUM_{T,N}(c, s⃗, ρ)
7:   else n has two children c_1 and c_2
8:     for each t⃗_1 ∈ ρ_{c_1} ⋉_{pred(n→c_1)} t⃗ do
9:       for each t⃗_2 ∈ ρ_{c_2} ⋉_{pred(n→c_2)} t⃗ do
10:        for each (s⃗_1, μ) ∈ ENUM_{T,N}(c_1, t⃗_1, ρ) do
11:          for each (s⃗_2, ν) ∈ ENUM_{T,N}(c_2, t⃗_2, ρ) do
12:            yield (s⃗_1 ∪ s⃗_2, μ × ν)
```

Proposition 5.2. *Let Q, T, and N be as above. If ρ is the T-reduct of db, then $\text{ENUM}_{T,N}(\rho)$ enumerates $Q(db)$.*

We now analyze the complexity of $\text{ENUM}_{T,N}$. First, observe that by definition of T-reducts, compatible tuples will exist at every node. 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 tuple is determined by the time taken to enumerate the tuples in $\rho_n \ltimes_{pred(p\to n)} \vec{t}$, where p is the parent of n. Since this is equivalent to $\pi_{var(n)}(\rho_n \bowtie_{pred(p\to n)} \vec{t})$ we can do this efficiently by creating an index on ρ_n by $(pred(p \to n), var(p), var(n))$. For example, in Section 4 we defined hash-maps and group-sorted GMRs so that given one tuple from a parent we could enumerate the compatible tuples in the child with constant delay. In general, the efficiency of enumeration will depend on the delay provided by the indices.

Proposition 5.3. *Assume that for every $n \in N$ with parent p we have an index on ρ_n by $(pred(p \to n), var(p), var(n))$ with delay f. Then, using these indices, $\text{ENUM}_{T,N}(\rho)$ enumerates $Q(db)$ with delay $O(|N| \times f(M))$ where M is given by $\max_{n\in N}(|\rho_n|)$. Therefore, the total time required to execute $\text{ENUM}_{T,N}(\rho)$ is $O(|Q(db)| \cdot f(M) \cdot |N|)$.*

In particular, if f is constant we obtain constant-delay enumeration of $|Q(db)|$. Recall that N is a constant under data complexity.

Update processing. To allow enumeration of $Q(db)$ under updates to db we need to maintain the T-reduct ρ (and, if present, its indexes) up to date. As illustrated in the previous section, it suffices to traverse the nodes of T in a bottom-up fashion. At each node n we have to compute the delta of ρ_n. For leaf nodes, this delta is given by u itself. For interior nodes, the delta can be computed from the delta and original reduct of its children. Algorithm 2 gives the pseudocode.

The fundamental part of Algorithm 2 is to compute joins and produce delta GMRs (Line 10), propagating updates from each node to its parent. When there is an update Δ_n to a node n with sibling m and parent p, we need to compute $\pi_{var(p)}\left(\rho_m \bowtie_{pred(p)} \Delta_n\right)$. To do this efficiently, we naturally store an index on ρ_m by $(pred(p), var(n), var(p))$. For example, we discussed how the hash-map I_{ρ_r} in Figure 4 plus the sorting on x of ρ_r allowed us to efficiently compute $\pi_{y,z,w}(\rho_r \bowtie_{x<z} \Delta\rho_s)$ on update $\Delta\rho_s$ to ρ_s.

Algorithm 2 Update(ρ, u)

Input: A T-reduct ρ for db and an update u.
Result: Transforming ρ to a T-reduc for $db + u$.
for each $n \in$ leafs(T) labeled by $r(\overline{x})$ **do**
  $\Delta_n \leftarrow u_{r(\overline{x})}$
for each $n \in$ nodes(T) $\setminus$ leafs(T) **do**
  $\Delta_n \leftarrow$ empty GMR over $var(n)$
for each $n \in$ nodes(T), traversed bottom-up **do**
  $\rho_n += \Delta_n$
  if n has a parent p and a sibling m **then**
    $\Delta_p += \pi_{var(p)} \left(\rho_m \bowtie_{pred(p)} \Delta_n\right)$
  else if n has parent p **then**
    $\Delta_p += \pi_{var(p)}(\Delta_n \mid pred(p))$

Summarizing the discussion above, to efficiently enumerate query results and process updates we need to store a T-reduct plus a set of indices on its GMRs. The data structure containing these elements is called a *T-representation*.

Definition 5.4 (T-representation)**.** Let Q be a free-connex acyclic GCQ, let T be a binary and strongly compatible GJT, and let db be a database for $at(Q)$. Let N be the sibling-closed connex subset of T such that $var(N) = out(Q)$. A *T-representation (T-rep for short)* of db is a data structure containing a T-reduct of db and, for each node n with parent p, the following set of indices:

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

Algorithms 1 and 2 plus the notion of T-representation provide a framework for dynamic query evaluation. By constructing the required T-reduct and set of indices (and their update procedures) one can automatically process free-connex acyclic GCQs under updates. Naturally, to implement such framework one needs to devise indices for a particular set of predicates. For example, Dyn is an instantiation of this framework to the class of NCQs, and in the previous section we intuitively showed how to instantiate this framework for GCQs based on equalities and inequalities. Next, we formally present the general set of indices required to process free-connex acyclic GCQs with equalities and inequalities.

IEDyn. For queries that have only equality and inequality predicates, the instantiation of a T-representation of db contains a T-reduct of db and, for each node n with parent p, the following data structures:

- For each $n \in N$, the index P_n on ρ_n from Definition 5.4 is obtained by doing two things. (1) First, we group ρ_n according to the variables in $var(n) \cap var(p)$. Then, per group, we sort the tuples according to the variables of $var(n)$ mentioned in $pred(p \to n)$ (if any).(2) We create a hash table that maps each tuple $\vec{t} \in \pi_{var(n)\cap var(p)}(\rho_n)$ to its corresponding group in ρ_n. In case that $var(n) \cap var(p)$ is empty, this hash table is omitted.
- For each node n with sibling m, the index S_n of Definition 5.4 is obtained by doing two things. (1) First, we group ρ_n according to the variables in $var(n) \cap var(m)$. Then, per group, we sort the tuples according to the variables of $var(n)$ mentioned in $pred(p)$ (if any). (2) We create a hash table that maps each tuple $\vec{t} \in \pi_{var(n)\cap var(m)}(\rho_n)$ to the corresponding group in $\vec{s} \in \rho_n$. In case that $var(n) \cap var(m)$ is empty, this hash table is omitted.

Given these data structures, we have described how to process the joins associated to P_n and S_n in Section 4. In Figure 4, I_{ρ_r} and I_{ρ_s} are examples of S_n, used for update propagation, while $I_{\rho_{\{y,z,w\}}}$ is an example of P_n, used for enumeration.

Note that the example query from Section 4 has at most one inequality between each pair of atoms. This causes each edge in T to consist of at most one inequality. As such, when creating the index P_n for a node $n \in N$, the reduct ρ_n will be sorted per group according to at most one variable. This is important for enumeration delay because, as exemplified in Section 4, we can then find compatible tuples by first identifying the corresponding group and then iterating over the sorted group from the start and stopping when the first non-compatible tuple is found. When there are multiple inequalities per pair of atoms then we will need to sort according to multiple variables under some lexicographic order. This causes enumeration delay to become logarithmic because compatible tuples will intermingle with non-compatible tuples and a binary search is necessary to find the next batch of compatible tuples in the group.

We denote by IEDyn the algorithm for processing free-connex acyclic GCQs with equalities and inequalities.

Theorem 5.5. *Let Q be a free-connex acyclic GCQ with equalities and inequalities only. Let T be a binary GJT strongly compatible with Q. Given a database db over $at(Q)$, a T-rep $\mathcal{D}$ of db and an update u,* IEDyn *transforms $\mathcal{D}$ into a T-rep of $db+u$ in time $O((|db| + |u|)\log(|db| + |u|))$. Also,* IEDyn *can enumerate $Q(db)$ with delay $O(\log(|db|))$ and, moreover, if every pair of atoms in Q has at most one inequality then this enumeration is done with constant delay.*

6. EXPERIMENTAL SETUP

In this section, we present the setup of our experimental evaluation, whose results are discussed in Section 7. We first present our practical implementation of IEDyn, then show the queries and update stream used for evaluation, and finally discuss the competing systems.

Practical Implementation. We have implemented IEDyn as a query compiler that generates executable code in the Scala programming language. The generated code instantiates a T-rep and defines *trigger functions* that are used for maintaining the T-rep under updates. Our implementation is basic in the sense that we use Scala off-the-shelf collection libraries (notably `MutableTreeMap`) to implement the required indices. Faster implementations with specialized code for the index structures are certainly possible.

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 will generate, 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 ρ_n and (2) the delta GMRs $\Delta\rho_n$ that are computed by Algorithm 2 when processing u. In this case, however, one also needs to index the $\Delta\rho_n$ similarly to ρ_n. In the *pull-based mode*, in contrast, the system only

Table 1: Queries for experimental evaluation.

Query	Expression
Q_1	$R(a,b,c) \bowtie S(d,e,f) \vert a < d$
Q_2	$R(a,b,c,k) \bowtie S(d,e,f,k) \vert a < d$
Q_3	$R(a,b,c) \bowtie S(d,e,f) \bowtie T(g,h,i) \vert a < d \wedge e < g$
Q_4	$R(a,b,c) \bowtie S(d,e,f) \bowtie T(g,h,i) \vert a < d \wedge d < g$
Q_5	$R(a,b,c,k) \bowtie S(d,e,f,k) \bowtie T(g,h,i) \vert a < d \wedge d < g$
Q_6	$R(a,b,c) \bowtie S(d,e,f,k) \bowtie T(g,h,i,k) \vert a < d \wedge d < g$
Q_7	$\pi_{a,b,d,e,f,g,h}(Q_4)$
Q_8	$\pi_{a,d,e,f,g,h,k}(Q_5)$
Q_9	$\pi_{d,e,f,g,h,k}(Q_6)$
Q_{10}	$\pi_{b,c,e,f,h,i}(Q_4)$
Q_{11}	$\pi_{b,c,e,f,h,i}(Q_5)$
Q_{12}	$\pi_{b,c,e,f,h,i}(Q_6)$

maintains the T-rep under updates but does not generate any output stream. Nevertheless, at any time a user can call the enumeration procedure to obtain the current result.

We have described in Section 5 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-rep for Q_F. When operating in push-based mode, for each update u, we use the T-representation 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 on T 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.

Queries and Streams. In contrast to the setting for equi-join queries where systems can be compared based on established benchmarks such as TPC-H and TPC-DS, there is no established benchmark suite for inequality-join queries.

We evaluate IEDyn on the GCQ queries listed in table 1. Here, queries Q_1–Q_6 are full join queries (i.e., queries without projections). Among these, Q_1, Q_3 and Q_4 are cross products with inequality predicates, while Q_2, Q_5 and Q_6 have at least one equality in addition to the inequality predicates. Queries Q_1 and Q_2 are binary join queries, while Q_3–Q_6 are multi-way join queries. Queries Q_7–Q_{12} project over the result of queries Q_4–Q_6. Among these, Q_7–Q_9 are free-connex acyclic while Q_{10}–Q_{12} are acyclic but not free-connex.

We evaluate these queries on streams of updates where each update consists of a single tuple insertion. The database is always empty when we start processing the update stream. We synthetically generate two kinds of update streams: *random-order* update streams and *temporally-ordered* update streams. In *random-order* 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. Random-order update streams are useful for comparing against systems that allow processing of out-of-order tuples; temporally-ordered update streams are useful for comparing against systems that assume events arrive always with increasing timestamp values. Examples of the latter are automaton-based CER systems.

A random update stream of size N for a query with k relations is generated as follows. First, we generate N/k tuples with random attribute values for each relation. Then, we insert tuples in the update stream by uniformly and randomly selecting them without repetitions. This ensures that there are N/k insertions from each relation in the stream. To utilize the same update stream for evaluating each system we compare to, each stream is stored in a file. We choose the values for equality join attributes uniformly at random from 1 to 200, except for the scalability and selectivity experiments in Section 7 where the interval depends on the stream size.

Temporally-ordered streams are generated similarly, but when a new insertion tuple is chosen, a new value is inserted in the attributes that are compared through inequalities. This value is larger than the corresponding values of previously inserted tuples. All attributes hold integer values, except for attributes c and i which contain string values.

Competitors. We compare IEDyn (IE) against DBToaster (DBT) [25], Esper (E) [18], SASE (SE) [2, 36, 39], Tesla (T) [13, 14], and ZStream (Z) [26] measuring *memory footprint*, *update processing time*, and *enumeration delay* as comparison metrics. The competing systems differ in their mode of operation (push-based vs pull-based) and some of them only support temporally-ordered streams.

DBToaster is a state-of-the-art implementation of HIVM. It operates in pull-based mode, and can deal with random-order update streams. DBToaster is particularly meticulous in that it materializes only useful views, and therefore it is an interesting implementation for comparison. DBToaster has been extensively tested on equi-join queries and has proven to be more efficient than a commercial database management system, a commercial stream processing system and an IVM implementation [25]. DBToaster compiles SQL statements into executable trigger programs in different programming languages. We compare against those generated in Scala from the DBToaster Release 2.2[3]. DBToaster uses actors[4] 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.

Esper is a CER engine with a relational model based on Stanford STREAM [3]. It is push-based, and can deal with random-order update streams. We use the Java-based open source[5] for our comparisons. Esper processes queries expressed in the Esper event processing language (EPL).

SASE 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[6]. This implementation does

[3]https://dbtoaster.github.io/
[4]https://doc.akka.io/docs/akka/2.5/
[5]http://www.espertech.com/esper/esper-downloads/
[6]https://github.com/haopeng/sase

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 [2].

Tesla/T-Rex is also an automaton-based CER system. It operates in push-based mode, and supports temporally-ordered update streams only. We use the publicly available C-based implementation[7]. This implementation operates in a publish-subscribe model where events are published by clients to the server (TRexServer). Clients can subscribe to receive recognized composite events. Tesla cannot deal with queries involving inequalities on multiple attributes such as 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 TRexServer from the start of the stream being processed until the end.

ZStream 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 of ZStream described in their original paper [26]. This paper does not describe how to treat projections, and as such we compare against ZStream only for full join queries Q_1–Q_6.

Due to space limitations, we cannot include the query expressions used for Esper (in EPL), SASE, and Tesla/T-Rex (rules) in this paper, but they are available at [21].

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++ base 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. EXPERIMENTAL EVALUATION

Before presenting our experimental results we make some remarks. First, when we compare against another system we run IEDyn in the operation mode supported by the competitor. For push-based systems we report the time required to both process the entire update stream and generate the changes to the output after each update. When comparing against a pull-based system, the measured time includes only processing the entire update stream. We later report the speed with which the result can be generated from the underlying representation of the output (a T-representation in the case of IEDyn). When comparing against a system that supports random-order update streams, we only report comparisons using streams of this type. We have also looked at temporally-ordered streams for these systems, but the throughput of the competing systems 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 for the sake of space.

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

Table 2: Maximum output sizes per query, k=1000.

Query	|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	2.7k	114,561k
Q_8	21k	411,669k
Q_9	21k	99,043k
Q_{10}	2.7k	114,561k
Q_{11}	21k	294,139k
Q_{12}	21k	297,873k

It is also important to remark that 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. Figure 5 compares the update processing time of IEDyn against the competing systems for full join queries Q_1–Q_6. We have grouped experiments that are run under comparable circumstances: in the top row experiments are conducted for push-based systems on temporally-ordered update streams (SE, T, Z); in the second row push-based systems on random-order update streams (E), and in the bottom row pull-based systems on random-order update streams (DBT). We observe that all of the competing systems have large processing times even for very small update stream sizes, and that for some systems execution even failed. All of these behaviors are due to the low selectivity of joins on this dataset. Table 7 shows the output size of each query for the largest stream sizes reported in Figure 5. We report on streams that generate outputs of different sizes below.

Figure 5 is complemented by Figures 6 and 7 where we plot the processing time and memory footprint used by IEDyn as a percentage of the corresponding usage in the competing systems. Both, SE and Z support temporally ordered streams, however, SE supports only queries Q_1, Q_2, and Q_4 and Z supports Q_1–Q_6, therefore in Figure 7 we show SE (right) and Z (left). Note that IEDyn significantly outperforms the competing systems on all full join queries. Specifically, it outperforms DBT up to one order of magnitude in processing time and up to two orders of magnitude in memory footprint. It outperforms T by up to two orders of magnitude in processing time, and more than one order of magnitude in memory footprint. Moreover, for these queries, even in 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.

Projections. Figure 6 shows that IEDyn significantly outperforms E and T on free-connex queries Q_7–Q_9: up to an order of magnitude improvement over the throughput of E and more than twofold improvement over that of T. Memory

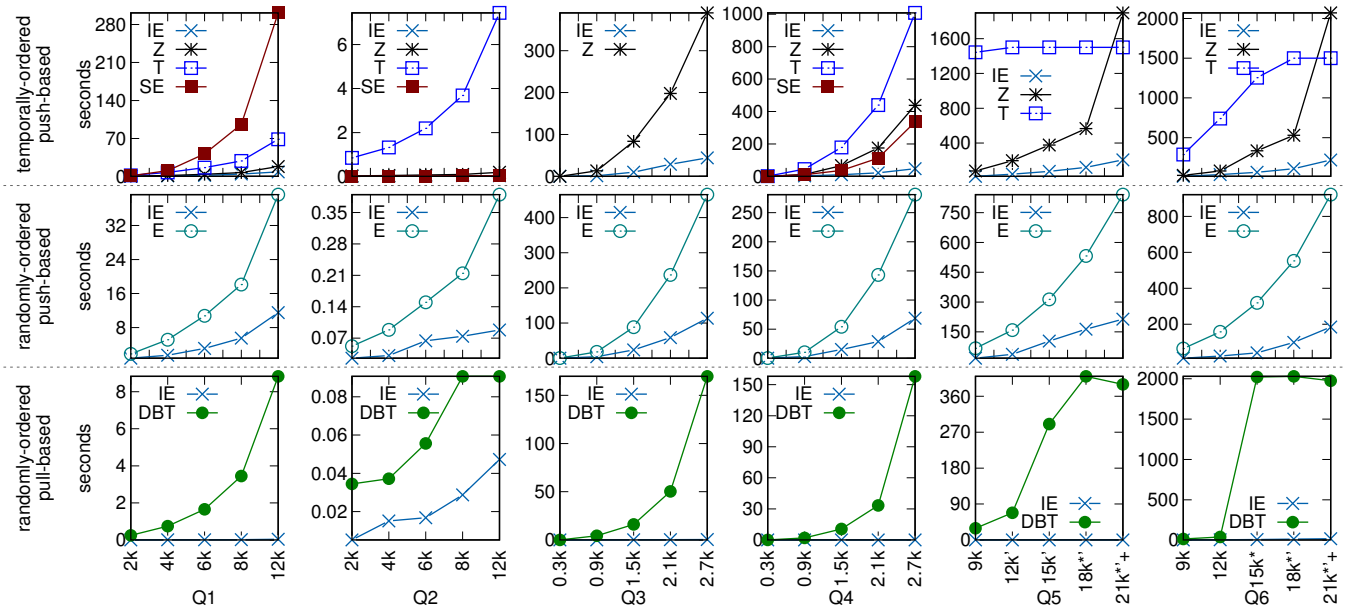

Figure 5: IEDyn **(IE) VS** (Z,DBT,E,T, SE) **on full join queries. The X-axis shows stream sizes and the y-axis update delay in** *seconds* **(*:** DBT **out of memory, +:** Z **out of memory,** $'$**:** T **was stopped after 1500 seconds).**

usage is also significantly less: one order of magnitude over E on the larger datasets for Q_7, and a consistent order of magnitude improvement over T. Similarly, IEDyn outperforms DBT on free-connex queries Q_7 and Q_8 in time and memory by one and two orders of magnitude, respectively.

For non-free-connex queries Q_{10}–Q_{12}, IEDyn continues to outperform E, T, and DBT in terms of processing time. In memory footprint IEDyn outperforms E for Q_{10} and Q_{12}. Compared to DBT, IEDyn still improves on memory footprint on non-free-connex queries, though less significantly. On the contrary, IEDyn largely improves memory usage over T on larger datasets, even on non-free-connex queries.

Result enumeration. We know from Section 5 that T-reps maintained by IEDyn feature constant delay enumeration (CDE). This theoretical notion, however, hides a constant factor that could decrease performance in practice when compared to full materialization. In Figure 8, 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 percentage 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 reported in Figures 5, 6, and 7.

Selective inequality joins. We have analyzed the performance of IEDyn over datasets that are uniformly distributed. On these datasets, inequality joins yield large query results. One could argue that this might not be realistic. To address this concern, we generate datasets with probability distributions 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.

Our results depicted in Figure 9 show that IEDyn not only outperforms existing systems on less selective inequality joins; we also perform better on very selective inequality joins consistently.

Scalability. To present that IEDyn performs consistently on varying sizes of input streams, we report the stream processing time and the memory footprint each time a 10% of the stream is processed in Figure 10. The x-axis shows the number of tuples (in millions) that have been processed. These results show that IEDyn has linearly increasing memory footprint as well as update delay as the stream size advances. We show results for queries Q_4, Q_5, Q_7, and Q_8 only due to space constraints.

8. DISCUSSION

In this paper, we have generalized the Dynamic Yannakakis (Dyn) algorithm, which was developped for the dynamic processing of acyclic aggregate-equijoin queries to an abstract framework called GDyn that efficiently processes GCQs with arbitrary θ-join predicates. To implement this framework for a particular set of predicates, suitable index structures need to be designed (cf. Definition 5.1). We have shown that for equality and inequality predicates ($<, \leq, =, \geq, >$) natural and efficient index structures based on hashing and sorting exist. We are currently investigating how disequalities ($\neq$) can be tackled in the GDyn framework. While it is relatively straightforward to obtain index structures that feature logarithmic delay, it is an open question whether indexes with constant delay exist. Other avenues of ongoing work include parallel and distributed versions of GDyn.

Acknowledgements M. Idris is supported by the Erasmus Mundus Joint Doctorate in "Information Technologies for Business Intelligence – Doctoral College (IT4BI-DC)". M. Ugarte is supported by the Brussels Captial Region–Innoviris (project SPICES). S. Vansummeren acknowledges gracious support from the Wiener-Anspach foundation.

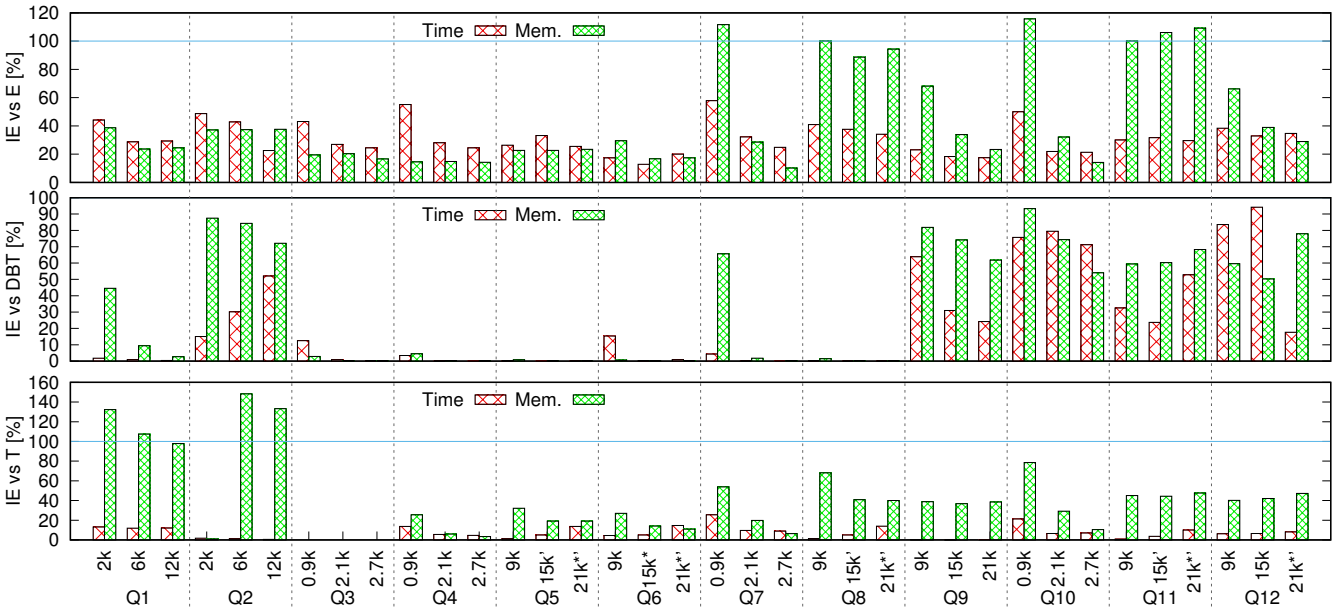

Figure 6: IEDyn **(IE) VS** (E, DBT, T) **fulljoin and projection queries, (*:** DBT **ran out of memory,** $'$**:** T **was stopped after 1500 seconds).**

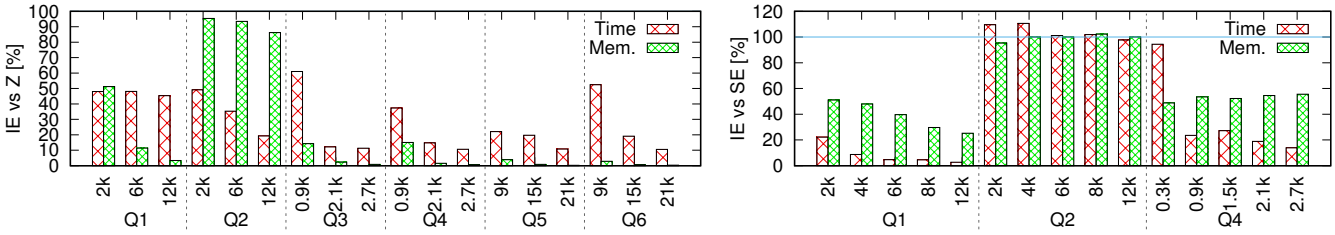

Figure 7: IEDyn **(IE) VS** SE **and** Z **on temporally ordered datasets.**

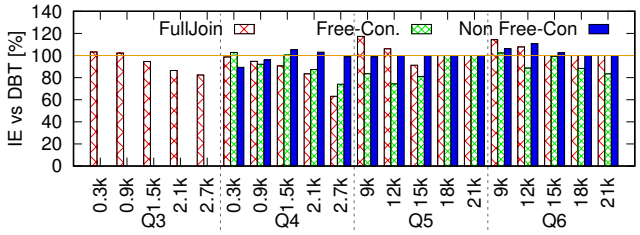

Figure 8: Enumeration of query results: IEDyn **vs** DBT**, different bars for** Q_4, Q_5, Q_6 **show their projected versions.**

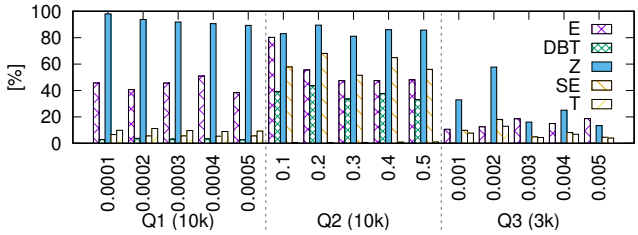

Figure 9: IEDyn **as percentage of** (E, DBT, SE, Z, T) **for higher join selectivities. X-axis shows queries with tuples per relation and selectivities.**

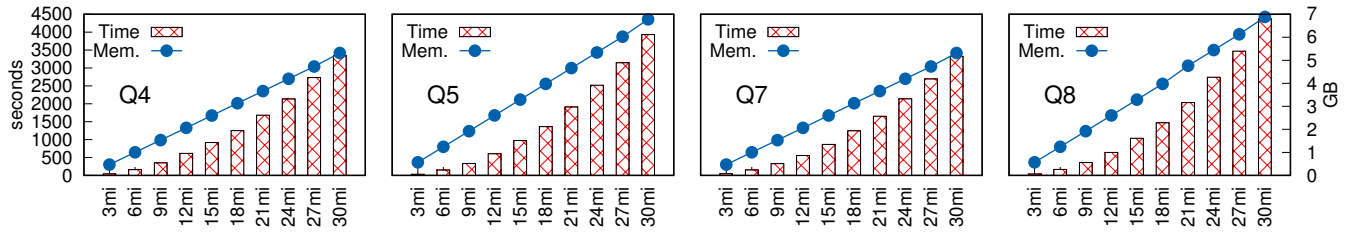

Figure 10: IEDyn **scalability** ($mi = 1,000,000$).

9. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[2] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proc. of SIGMOD*, pages 147–160, 2008.

[3] 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.

[4] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Proc. of CSL*, pages 208–222, 2007.

[5] N. Bakibayev, T. Kočiský, D. Olteanu, and J. Závodný. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013.

[6] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering conjunctive queries under updates. In *Proc. of PODS*, pages 303–318, 2017.

[7] P. A. Bernstein and N. Goodman. The power of inequality semijoins. *Inf. Syst.*, 6(4):255–265, 1981.

[8] J. Brault-Baron. *De la pertinence de l'énumération: complexité en logiques*. PhD thesis, Université de Caen, 2013.

[9] 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 *SIGMOD*, pages 1100–1102, 2007.

[10] A. Buchmann and B. Koldehofe. Complex event processing. *IT-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 2009.

[11] T. Cormen. *Introduction to Algorithms, 3rd Edition:*. MIT Press, 2009.

[12] G. Cormode, S. Muthukrishnan, and W. Zhuang. Conquering the divide: Continuous clustering of distributed data streams. In *ICDE*, pages 1036–1045, 2007.

[13] G. Cugola and A. Margara. TESLA: a formally defined event specification language. In *Proc. of DEBS*, pages 50–61, 2010.

[14] G. Cugola and A. Margara. Complex event processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012.

[15] 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.

[16] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *VLDB Conference*, pages 443–452, 1991.

[17] J. Enderle, M. Hampel, and T. Seidl. Joining interval data in relational databases. In *Proc. SIGMOD*, pages 683–694, 2004.

[18] EsperTech. Esper complex event processing engine. `http://www.espertech.com/`.

[19] M. P. Groover. *Automation, production systems, and computer-integrated manufacturing*. 2007.

[20] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB Conference*, pages 562–573, 1995.

[21] M. Idris. Queries in sase, tesla, esper, and sql for dbt/iedyn expression. `http://cs.ulb.ac.be/~midris/iedyn.html`.

[22] M. Idris, M. Ugarte, and S. Vansummeren. The dynamic Yannakakis algorithm: Compact and efficient query processing under updates. In *Proc. of SIGMOD*, 2017.

[23] 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.

[24] C. Koch. Incremental query evaluation in a ring of databases. In *Proc. of PODS*, pages 87–98, 2010.

[25] 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.

[26] Y. Mei and S. Madden. ZStream: a cost-based query processor for adaptively detecting composite events. In *Proc. SIGMOD*, pages 193–206, 2009.

[27] M. Nikolic, M. Dashti, and C. Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proc.of SIGMOD*, pages 511–526, 2016.

[28] M. Nikolic and D. Olteanu. Incremental view maintenance with triple lock factorisation benefits. In *SIGMOD 2018*, 2018. To appear.

[29] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *ACM TODS*, 40(1):2:1–2:44, 2015.

[30] B. Sahay and J. Ranjan. Real time business intelligence in supply chain analytics. *Information Management & Computer Security*, 16(1):28–48, 2008.

[31] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *Proc. of SIGMOD*, pages 3–18, 2016.

[32] N. P. Schultz-Møller, M. Migliavacca, and P. R. Pietzuch. Distributed complex event processing with query rewriting. In *Proc. of DEBS*, 2009.

[33] L. Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Record*, 44(1):10–17, 2015.

[34] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, (4):42–47, 2005.

[35] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *Proc. of STOC*, pages 137–146, 1982.

[36] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proc. of SIGMOD*, pages 407–418, 2006.

[37] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB Conference*, pages 82–94, 1981.

[38] M. Yoshikawa and Y. Kambayashi. Processing inequality queries based on generalized semi-joins. In *VLDB Conference*, pages 416–428, 1984.

[39] H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in complex event processing. In *Proce. of SIGMOD*, 2014.