The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates

Muhammad Idris
Université Libre de Bruxelles
and TU Dresden
muhammad.idris@ulb.ac.be

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

Stijn Vansummeren
Université Libre de Bruxelles
stijn.vansummeren@ulb.ac.be

ABSTRACT

Modern computing tasks such as real-time analytics require refresh of query results under high update rates. Incremental View Maintenance (IVM) approaches this problem by materializing results in order to avoid recomputation. IVM naturally induces a trade-off between the space needed to maintain the materialized results and the time used to process updates. In this paper, we show that the full materialization of results is a barrier for more general optimization strategies. In particular, we present a new approach for evaluating queries under updates. Instead of the materialization of results, we require a data structure that allows: (1) linear time maintenance under updates, (2) constant-delay enumeration of the output, (3) constant-time lookups in the output, while (4) using only linear space in the size of the database. We call such a structure a Dynamic Constant-delay Linear Representation (DCLR) for the query. We show that Dyn, a dynamic version of the Yannakakis algorithm, yields DCLRs for the class of free-connex acyclic CQs. We show that this is optimal in the sense that no DCLR can exist for CQs that are not free-connex acyclic. Moreover, we identify a sub-class of queries for which Dyn features constant-time update per tuple and show that this class is maximal. Finally, using the TPC-H and TPC-DS benchmarks, we experimentally compare Dyn and a higher-order IVM (HIVM) engine. Our approach is not only more efficient in terms of memory consumption (as expected), but is also consistently faster in processing updates.

KEYWORDS

Incremental View Maintenance; Dynamic query processing; Acyclic joins

1. INTRODUCTION

Real-time analytics find applications in Financial Systems , Industrial Control Systems, Business Intelligence and Online Machine Learning, among many others (see [15] for a survey). Generally, the analytical results that need 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 query evaluation*, where a given query Q has to be evaluated against a database that is constantly updated. In this setting, 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 has traditionally been approached from Incremental View Maintenance (IVM) [13]. IVM techniques materialize $Q(db)$ and evaluate *delta queries*. These take as input db, u and the materialized $Q(db)$, and return the set of tuples to add/delete from $Q(db)$ 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. Research in this area has recently received a big boost with the introduction of Higher-Order IVM (HIVM) [25, 26, 30]. Given a query Q, HIVM not only defines the delta query ΔQ, but also materializes it. Mo reover, it defines higher-order delta queries (i.e., delta queries for delta queries, denoted $\Delta^2 Q, \Delta^3 Q, \dots$), where every $\Delta^j Q$ describes how the materialization of $\Delta^{j-1} Q$ should change under updates. This method is highly efficient in practice, and is formally in a lower complexity class than IVM [25].

(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 than IVM since it not only materializes the result of Q but also the results of the higher-order delta queries. Second, IVM and HIVM only exploit the information provided by the materialized views to process updates, while additional forms of information could result in better update rates. Consider for example the query $Q = R(A, B) \bowtie S(B, C)$ and a database with N tuples in R and N tuples in S, all with the same B value. The materialization of $Q(db)$ in this case uses $\Theta(N^2)$ space and is useless for re-computing Q under updates. In contrast, a simple index on B for R and S would allow for efficient enumeration of the set of tuples that need to be added/removed from $Q(db)$ to obtain $Q(db + u)$. It is important to note that even for queries whose result is smaller than the database, aggressive materialization of higher-order delta queries in HIVM can still cause these problems to appear. Indeed, some higher-order delta queries are partial join results, which can be larger than both db and $Q(db)$.

While these problems are inherent to (H)IVM methods based on materialization, they can be avoided by taking a

SIGMOD '17 May 14–19, 2017, Chicago, IL, USA

ISBN 978-1-4503-4197-4/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3035918.3064027

different approach to dynamic query evaluation: instead of maintaining $Q(db)$, we could maintain a data structure from which $Q(db)$ can be *generated* as efficiently as if it were materialized. This notion is formalized by the theoretical database community by requiring that the output $Q(db)$ can be enumerated from the data structure *with constant delay* [5]. Intuitively, data structures that feature constant-delay enumeration (CDE for short) are aimed at representing data in compressed form yet have a streaming decompression algorithm that can spend only a constant amount of work to produce each new output tuple [37].

While there is increasing work on query evaluation with constant-delay enumeration [5, 7, 10, 31, 37], known results either present (involved) theoretical algorithms, or study the static setting without updates. In this paper, therefore, we are concerned with designing a practical algorithm for dynamic query evaluation based on constant-delay enumeration. In particular, to dynamically process a query Q we desire a *Dynamic Constant-delay Linear Representation (DCLR)* of Q, meaning that for every database db we can compute a data structure $\mathcal{D}_{db}$ with the following properties:

- (P_1) $\mathcal{D}_{db}$ allows to enumerate $Q(db)$ with constant delay.
- (P_2) For any tuple $\vec{t}$, we can use $\mathcal{D}_{db}$ to check whether $\vec{t} \in Q(db)$ in constant time.
- (P_3) $\mathcal{D}_{db}$ requires only $O(\| db \|)$ space. As such $\mathcal{D}_{db}$ depends only on db and is independent of the size of $Q(db)$.
- (P_4) $\mathcal{D}_{db}$ features efficient maintenance under updates: given $\mathcal{D}_{db}$ and update u to database db, we can compute $\mathcal{D}_{db+u}$ in time $O(\| db \| + \| u \|)$. In contrast, both IVM and HIVM may require $\Omega(\|u\| + \|Q(db+u)\|)$ time in the worst case.

It is important to note that we consider query evaluation in main memory and measure time and space under data complexity [38]. 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.

Our contributions are as follows. We focus on the class of conjunctive aggregate queries (CAQ) evaluated under multiset semantics. Conjunctive aggregate queries are queries that compute aggregates (e.g., SUM, AVG, ...) over the result of a conjunctive query (CQ, also known as select-project-join query). As a first contribution we discuss how to modify the Yannakakis algorithm [39] to obtain a static query evaluation algorithm that satisfies properties P_1 and P_3 for the restricted class of acyclic join queries. We call this variant CDY, for Constant Delay Yannakakis. We then introduce Dyn, a dynamic version of CDY, and show that it yields DCLRs (properties P_1–P_4) for the acyclic join queries. Dyn is a good algorithmic core to build practical dynamic algorithms on, for the following reasons.

(1) Like standard Yannakakis, Dyn is a conceptually simple algorithm, and therefore easy to implement.

(2) We show that Dyn can support not only join queries (which do not allow projection), but also CQs (with projection) that belong to the class of *free-connex acyclic* CQs. This is optimal, in the sense that results by Bagan et al. [5] and Brault-Baron [10] for the static setting imply that, under certain complexity-theoretic assumptions, a DCLR can exist for a CQ Q only if Q is free-connex acyclic. In other words, Dyn is able to evaluate the most general subclass of conjunctive queries satisfying P_1–P_4.

(3) Furthermore, in very recent work Berkholz et al [7] have characterized the class of self-join free CQs that feature CDE and that can be maintained in $O(1)$ time under single-tuple updates. They show that this class corresponds to the class of so-called *q-hierarchical* queries, a strict subclass of the free-connex acyclic queries. We match their lower bound: for (not necessarily self-join free) q-hierarchical CQs the Dyn algorithm processes single-tuple updates in constant time. For non q-hierarchical queries, Berkholz et al.'s result yields it unlikely that single-tuple updates can be processsed in constant time. While for such queries, Dyn hence naturally also requires more processing time, our experiments show that it remains highly effective.

(4) For single-tuple updates, Dyn also allows us to enumerate the delta result $Q(db+u)-Q(db)$ with constant delay. This result is relevant for *push-based* query processing systems, where users do not ping the system for the complete current query answer, but instead ask to be notified of the changes to the query results when the database changes.

Building on Dyn, we present an extended algorithm that allows for dynamic evaluation of acyclic CAQs. In particular, for an CAQ Q whose join Q' is acyclic but not free-connex, we obtain a dynamic query processing method that is based on delta-enumeration of a free-connex projection of Q' to materialize the resulting aggregates $Q(db)$. We hence require $O(\| db \| + \|Q(db)\|)$ memory in this case, just like IVM methods.

Finally, we experimentally compare our approach against HIVM on the industry-standard benchmarks TPC-H and TPC-DS. Our experiments show that, for the class of acyclic CAQs, our method is up to one order of magnitude more efficient than HIVM, both in terms of update time and memory consumption. At the same time, our experiments show that the enumeration of $Q(db)$ from $\mathcal{D}_{db}$ is as fast (and sometimes, even faster) as when $Q(db)$ was materialized as an array.

Organization. This paper is further organized as follows. We discuss additional related work in Section 2 and introduce background concepts in Section 3. Dyn is developed in Section 4 and experimentally evaluated in Section 5. Because of space constraints, full proofs are deferred to the full version of this paper, but cruxes of some key results may be found in the Appendix.

2. RELATED WORK

Incremental View Maintenance. The problem of incrementally maintaining materialized answers to conjunctive queries under updates has been extensively studied [3, 8, 9, 11, 12, 20, 26, 30, 33], and has been adapted to support different types of aggregates [28, 35]. A natural extension of IVM is the maintenance of *auxiliary* views [24, 34], which have been also adapted to allow for aggregate queries [21]. IVM has been influential to several other areas of databases (see [13] for a recent survey). Our work differs from IVM in that we maintain data structures that do not fully materialize query results.

Factorized Databases. *Factorized Databases* are ingenious succinct representations of relational tables [31]. They allow for constant-delay enumeration and do not only reduce memory consumption, but can also avoid redundancy and

speed up query processing [6,31,36]. While factorized query evaluation is not limited to acyclic queries (as we are), it has only been studied in the static setting without updates.

Join algorithms. The well-known Yannakakis algorithm evaluates acyclic join queries in $O(\|db\| + \|Q(db)\|)$ by using a join tree of such query [39]. Worst-case optimal algorithms have been developed for more general classes of queries by inspecting other forms of query decompositions (see [29] for a survey). Recently, join algorithms derived from query decompositions have been identified to use intermediate data structures similar to factorized databases [31]. These data structures, however, are designed for the static setting and not to react to updates. Recent work has also extended known join algorithms to allow for multiple aggregations on top of join queries [2, 23] in the static setting.

The Generalized Distributed Law. The Generalized Distributed Law (GDL) is an algorithm for solving the *marginalize a product function* (MPF) problem [4]. It has been recently shown to be equivalent to algorithms for computing aggregate-join queries with one aggregate [23]. The algorithm DYN developed in this paper can be seen as a strategy for solving the MPF problem under the dynamic setting.

3. PRELIMINARIES

We adopt the data model of Generalized Multiset Relations (GMRs for short) [25, 26]. A GMR is a relation in which each tuple is associated to an integer in $\mathbb{Z}$. Figure 1 shows several examples. Note that in a GMR, in contrast to classical multisets, the multiplicity of a tuple can be negative. This allows to treat insertions and deletions symmetrically, as we will later see. To avoid ambiguity, we give a formal definition of GMRs that will be used throughout this paper.

Tuples. We first introduce some notation for tuples. Let $\overline{x}$ be a set of *variables* (also commonly known as *column names* or *attributes*). We write $\mathbb{T}[\overline{x}]$ for the universe of all possible tuples over $\overline{x}$. If $\vec{t} \in \mathbb{T}[\overline{x}]$ and y is a variable in $\overline{x}$ then we write $\vec{t}(y)$ for the value assigned to y by $\vec{t}$. If $\overline{y} \subseteq \overline{x}$ then we write $\vec{t}[\overline{y}]$ for the tuple over $\overline{y}$ obtained from $\vec{t}$ by removing all variables in $\overline{x} \setminus \overline{y}$. For example, if $\vec{t} = \langle A\colon 5, B\colon 4, C\colon 3\rangle$ then $\vec{t}(A) = 5$ and $\vec{t}[B, C] = \langle B\colon 4, C\colon 3\rangle$.

GMRs. A *generalized multiset relation* (GMR) over $\overline{x}$ is a function $R\colon \mathbb{T}[\overline{x}] \to \mathbb{Z}$ from relation tuples over $\overline{x}$ to integers. Every GMR R is a *total* function from the (possibly infinite) set $\mathbb{T}[\overline{x}]$ to $\mathbb{Z}$ and hence, conceptually, is an infinite object. However, every GMR is required to have finite *support* $\text{supp}(R) := \{\vec{t} \in \mathbb{T}[\overline{x}] \mid R(\vec{t}) \neq 0\}$. Intuitively, $R(\vec{t}) = 0$ indicates that $\vec{t}$ is absent from R. The fact that R must have finite support indicates that R is a finite relation. To illustrate, in Figure 1, $R(\langle a, b\rangle) = 2$, hence present, while $R(\langle a, b'\rangle) = 0$, hence absent (and not shown). In what follows, we abuse notation and write $(\vec{t}, \mu) \in R$ to indicate that $\vec{t} \in \text{supp}(R)$ and $R(\vec{t}) = \mu$; $\vec{t} \in R$ to indicate $\vec{T} \in \text{supp}(R)$; and $|R|$ for $|\text{supp}(R)|$. We say that R is empty if $\text{supp}(R) = \emptyset$. The set of all GMRs over $\overline{x}$ is denoted by $\mathbb{GMR}[\overline{x}]$. A GMR is *positive* if $R(\vec{t}) > 0$ for all $\vec{t} \in \text{supp}(R)$.

Operations on GMRs. Let R and S be GMRs over $\overline{x}$, T a GMR over $\overline{y}$, and $\overline{z} \subseteq \overline{x}$. The operations union $(R + S)$, minus $(R - S)$, join $(R \bowtie T)$ and projection $(\pi_{\overline{z}} R)$ over GMRs are defined as follows.

$$\begin{array}{lll} R + S & \in \mathbb{GMR}[\overline{x}] & : \vec{t} \mapsto R(\vec{t}) + S(\vec{t}) \\ R - S & \in \mathbb{GMR}[\overline{x}] & : \vec{t} \mapsto R(\vec{t}) - S(\vec{t}) \end{array}$$

R

A	B	$\mathbb{Z}$
a	b	2
a'	b'	3

S

A	B	$\mathbb{Z}$
a	b	5
a	b'	4

T

A	C	$\mathbb{Z}$
a	c	4
a	b'	5

$\pi_A(S)$

A	$\mathbb{Z}$
a	9

$R + S$

A	B	$\mathbb{Z}$
a	b	7
a'	b'	3
a	b'	4

$R - S$

A	B	$\mathbb{Z}$
a	b	-3
a'	b'	3
a	b'	-4

$R \bowtie T$

A	B	C	$\mathbb{Z}$
a	b	c	8
a	b	c'	15

Figure 1: Operations on GMRs

$$\begin{array}{lll} R \bowtie T & \in \mathbb{GMR}[\overline{x} \cup \overline{y}] & : \vec{t} \mapsto R(\vec{t}[\overline{x}]) \times S(\vec{t}[\overline{y}]) \\ \pi_{\overline{z}} R & \in \mathbb{GMR}[\overline{z}] & : \vec{t} \mapsto \sum_{\vec{s} \in \mathbb{T}[\overline{x}], \vec{s}[\overline{z}] = \vec{t}} R(\vec{s}) \end{array}$$

Figure 1 illustrates these operations. Note that GMRs there are positive, modeling standard multisets. Hence union, join, and projection correspond to the classical operations from relational algebra under multiset (i.e., bag) semantics. Minus is not relational difference, since it simply subtracts multiplicities (notice this could yield negative multiplicities).

Query Language. Conjunctive Queries (CQs) are expressions of the form

$$Q = \pi_{\overline{y}} \left(r_1(\overline{x_1}) \bowtie \cdots \bowtie r_n(\overline{x_n}) \right).$$

Here, $r_1, \ldots, r_n$ are *relation symbols*; $\overline{x_1}, \ldots, \overline{x_n}$ are sets of variables, and $\overline{y} \subseteq \overline{x_1} \cup \cdots \cup \overline{x_n}$ is the set of *output variables*, also denoted by $out(Q)$. If $\overline{y} = \overline{x_1} \cup \cdots \cup \overline{x_n}$ then Q is a *join query* and simply denoted as $r_1(\overline{x_1}) \bowtie \cdots \bowtie r_n(\overline{x_n})$. The pairs $r_i(\overline{x_i})$ are called *atomic queries* (or simply *atoms*).

A *database* over a set A of atoms is a function db that maps every atom $r(\overline{x}) \in A$ 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: substitute each atom $r(\overline{x})$ in Q by $db_{r(\overline{x})}$, and subsequently apply the operations according to the structure of Q.

Discussion. For ease of notation in the rest of the paper, we have not included relational selection $\sigma_\theta(r(\overline{x}))$ in queries. This is without loss of generality, as to dynamically process a Select-Project-Join query we can always filter out irrelevant tuples. For example, for $Q = \pi_{\overline{z}} \left(\sigma_{\theta_1}(r(\overline{x})) \bowtie \sigma_{\theta_2}(s(\overline{y})) \right)$ we can consider new relation symbols r' and s' and dynamically process $Q' = \pi_{\overline{z}}(r'(\overline{x}) \bowtie s'(\overline{y}))$ instead. Then, whenever r and/or s are updated, it suffices to *discard* the tuples that do not satisfy the corresponding filter, and propagate the rest of the updates to relations r' and s' to update Q'.

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. 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 $Q(db + u)$.

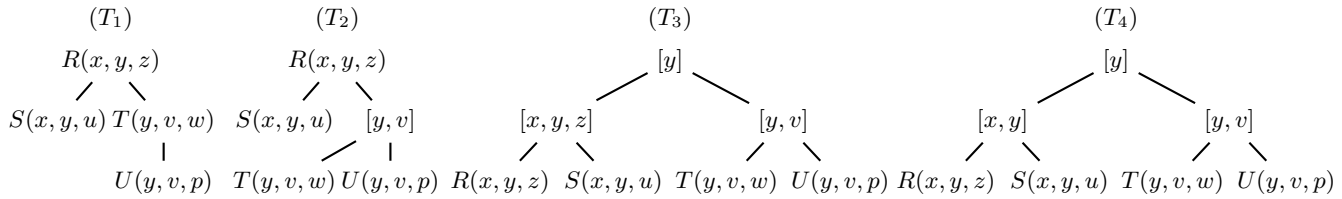


Figure 2: Width-one GHDs for $\{R(x,y,z), S(x,y,u), T(y,v,w), U(y,v,p)\}$. T_1 is a traditional join tree, T_3 and T_4 are generalized join trees. In addition, T_4 is simple.

3.1 Acyclicity

Throughout the paper we focus on the class of *acyclic* queries. While there are many equivalent ways of defining acyclic queries [1] we will use here a characterization of the acyclic queries in terms of those queries that have a Generalized Hypertree Decomposition (GHD for short) of width one [19]. Width-one GHDs generalize traditional join trees [1] by also allowing partial hyperedges to occur as nodes in the tree. Intuitively, these partial hyperedges represent projections of single atoms. The importance of this feature will become clear at the end of Section 4, where we show the existence of acyclic queries for which traditional join trees (where only full hyperedges can occur) do not induce optimal complexity algorithms under the setting of dynamic query evaluation.

To simplify notation, we denote the set of all variables (resp. the set of all atoms) that occur in a mathematical object X (such as a query) by $\text{var}(X)$ (resp. $at(X)$). In particular, if X is itself a set of variables, then $\text{var}(X) = X$.

Definition 3.1 (Width-1 GHD)**.** Let A be a finite set of atoms. A *hyperedge in* A is a set $\overline{x}$ of variables such that $\overline{x} \subseteq \text{var}(\boldsymbol{a})$ for some atom $\boldsymbol{a} \in A$. We call $\overline{x}$ *full* in A if $\overline{x} = \text{var}(\boldsymbol{a})$ for some $\boldsymbol{a} \in A$, and partial otherwise. A *Generalized Hypertree Decomposition (GHD) of width one* for A is a directed tree $T = (V, E)$ such that:[1]

- All nodes of T are either atoms or hyperedges in A. Moreover, every atom in A occurs in T.
- Whenever the same variable x occurs in two nodes m and n of T, then x occurs in each node on the unique undirected path linking m and n.

If all nodes in T are atoms, then T is a *traditional join tree*.

To illustrate, Figure 2 shows four width-one GHDs for $\{R(x,y,z), S(x,y,u), T(y,v,w), U(y,v,p)\}$. T_1 is traditional while the others are not.

Definition 3.2 (Acyclicity)**.** A CQ Q is *acyclic* if there exists a width-one GHD T for $at(Q)$, and is cyclic otherwise.

For example the width-one GHDs of Figure 2 show that $R(x,y,z) \bowtie S(x,y,u) \bowtie T(y,v,w) \bowtie U(y,v,p)$ is acyclic. In contrast, $R(x,y) \bowtie S(y,z) \bowtie T(x,z)$, the triangle query, is the prototypical cyclic join query.

For the rest of the paper, it will be convenient to focus on width-one GHDs of a particular form. We call such restricted GHDs *generalized join trees*.

Definition 3.3 (Generalized Join Tree)**.** A *generalized join tree* for set of atoms A is a width-one GHD T for A in which all atoms occur as leafs. Moreover, every interior node n must have at least one child c such that $\text{var}(n) \subseteq \text{var}(c)$.

[1]Readers familiar with the usual definition of GHDs of arbitrary width may find a discussion of the correspondence between our definition and the usual one in Appendix A.

In Figure 2, trees T_3 and T_4 are generalized join trees; trees T_1 and T_2 are not. The following proposition (proof in Appendix A) shows that we may restrict our attention to generalized join trees without loss of generality.

Proposition 3.4. *If there exists a width-one GHD for set of atoms A, then there also exists a generalized join tree for A. Consequently, a CQ Q is acyclic iff $at(Q)$ has a generalized join tree.*

In what follows, we will refer to generalized join trees simply as join trees. When n is a node of a join tree T, c is a child of n, and $\text{var}(n) \subseteq \text{var}(c)$, we call c a *guard* of n. By definition, there is a guard for every hyperedge. We denote by $\text{grd}(n)$ the set of guards of n, by $\text{ch}(n)$ the set of children of n, and by $\text{ng}(n)$ the set $\text{ch}(n) \setminus \text{grd}(n)$ of *non-guards* of n. Finally, we define $\text{pvar}(c)$ to be the set of variables that c has in common with its parent ($\text{pvar}(c) = \emptyset$ for the root). For example, in T_3 of Figure 2, $\text{pvar}(S(x,y,u)) = \{x,y\}$.

3.2 Computational Model

We focus on dynamic query evaluation in main-memory and analyze performance under data complexity [38]. We assume a model of computation where tuple values and integers take $O(1)$ space and arithmetic operations on integers as well as memory lookups are $O(1)$ operations. We further assume that every GMR R can be represented by a data structure that allows (1) enumeration of R with constant delay (as defined in Section 4.1); (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 size that is proportional to the number of tuples in the support of R. We further assume the existence of dynamic data structures that can be used to index GMRs on a subset of their variables. Concretely if R is a GMR over $\overline{x}$ and I is an index of R on $\overline{y} \subseteq \overline{x}$ then we assume that for every $\overline{y}$-tuple $\vec{s}$ we can retrieve in $O(1)$ time a pointer to the GMR $I(\vec{s}) \in \mathbb{GMR}[\overline{x}]$ consisting of all tuples that project to $\vec{s}$, as formally defined by

$$I(\vec{s}) \in \mathbb{GMR}[\overline{x}] \colon \vec{t} \mapsto \begin{cases} R(\vec{t}) & \text{if } \vec{t}[\overline{y}] = \vec{s} \\ 0 & \text{otherwise} \end{cases}$$

Moreover, we assume that single-tuple insertions and deletions to R can be reflected in the index in $O(1)$ time and that an index takes space linear in the support of R. Essentially, our assumptions amount to perfect hashing of linear size [14]. Although this is not realistic for practical computers [32], it is well known that complexity results for this model can be translated, through amortized analysis, to average complexity in real-life implementations [14].

4. DYNAMIC YANNAKAKIS

In this section we develop Dyn, a dynamic version of the Yannakakis algorithm. In Section 4.1, we introduce the notion of constant-delay enumeration. Then, in Section 4.2 we show that, for acyclic join queries, a representation satisfying properties P_1 and P_3 of the Introduction can be obtained by slightly modifying the Yannakakis algorithm. We introduce Dyn in Section 4.3, and show in Sections 4.4- 4.5 that this also gives DCLRs for CQs that are free-connex acyclic. We show that this is optimal in two distinct ways in Section 4.6.

4.1 Constant delay enumeration

Definition 4.1. 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. Moreover, D supports *constant-delay enumeration* (CDE) of E if, when ENUM(D) is invoked, the time until the output of the first element; the time between any two consecutive elements; and the time between the output of the last element and the termination of ENUM(D), are all constant. In particular, these times cannot depend on the size of D nor on the size of E. We say that D supports constant-delay enumeration of a GMR R if D supports constant-delay enumeration of the set $E_R = \{(\vec{t}, R(\vec{t})) \mid \vec{t} \in \text{supp}(R)\}$.

As a trivial example of CDE of a GMR R, assume that the pairs $(\vec{t}, R(\vec{t}))$ of E_R are stored in an array A (without duplicates). Then A supports CDE of R: ENUM(A) simply iterates over each element in A, one by one, always outputting the current element. To see that this is correct, first observe that all pairs of E_R will be output exactly once. Moreover, the time required to output the first pair is the time required to fetch the first array element, hence constant. Similarly, the time required to produce each subsequent output tuple is the time required to fetch the next array element, again constant. Finally, checking whether we have reached the end of E_R amounts to checking whether we have reached the end of the array, again taking constant time.

This example shows that in order to do CDE of the result $Q(db)$ of a query Q on input database db, we can always (naively) materialize $Q(db)$ in an in-memory array A. Unfortunately, A then requires memory proportional to $\|Q(db)\|$ which, depending on the query, 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 A: namely, with constant delay. The key idea to obtain this is *delayed evaluation*. To illustrate this, consider that we are asked to compute the Cartesian product of R and S. Then it suffices to simply store R and S, requiring $O(\|R\| + \|S\|) = O(\|db\|)$ memory. To enumerate $R \times S$, ENUM simply executes a nested-loop based Cartesian product over R and S. This satisfies the properties of CDE. Indeed, every element of $R \times S$ will be output exactly once. Moreover, the time required to output the first element of $R \times S$ is the time required to initialize a pointer to the first elements of R and S (hence constant). The time required to produce each subsequent element is bounded by the time required to either advance the pointer in S, or advance the pointer in R and reset the pointer in S to the beginning. In both cases, this is constant. Finally, checking whether we have reached the end of $R \times S$ again takes constant time.

The situation becomes more complex for queries that involve joins instead of Cartesian products. Consider for example the query $Q = R(A, B) \bowtie S(B, C)$. Simply delaying evaluation does not yield constant-delay enumeration. Indeed, suppose that we evaluate Q using a simple in-memory hash join with R as build relation and S as probe relation. Assume that the corresponding index of R on B (i.e. the hash table) has already been computed. When iterating over S to probe the hash table, we may have to visit an unbounded number of S-tuples that do not join with any of the R-tuples. Consequently, there is no constant that bounds the delay between consecutive outputs. A similar analysis shows that other join algorithms, such as the sort-merge join, do not yield enumeration with constant delay.

In essence, therefore, a data structure that allows CDE of $Q(db)$ must be able to produce all output tuples and their multiplicities without spending any extra time in building auxiliary data structures to help in enumeration (such as hash tables or sorted versions of the input relations), nor can it afford to waste time in processing input tuples that in the end do not appear in $Q(db)$.

How do we obtain CDE for $R(A, B) \bowtie S(B, C)$? Intuitively speaking, if in our hash join algorithm we can ensure to only iterate over those S-tuples that have matching R-records, we trivially obtain a CDE algorithm. In a broader sense, we need to maintain under updates, for the relations that are used as *probe* relations, the set of tuples that will match the corresponding build relation(s). We call these tuples the *live tuples*. In the following sections we gradually devise a more general algorithm that follows this idea. Intuitively, this algorithm dynamically maintains the hash tables and the live values for a query in a DCLR.

4.2 Constant Delay Yannakakis

Acyclic full join queries are evaluated in $O(\|db\| + \|Q(db)\|)$ time by the well-known Yannakakis algorithm. For future reference, we recall the operation of the Yannakakis algorithm [39], formulated in our setting. We first need to introduce the semi-join operation for GMRs.

Definition 4.2. The semijoin $R \ltimes S$ of a GMR $R[\overline{x}]$ by a GMR S is the GMR over $\overline{x}$ defined by

$$R \ltimes S \in \mathbb{GMR}[\overline{x}] \colon \vec{s} \mapsto \begin{cases} R(\vec{s}) & \text{if } \vec{s} \in \pi_{\overline{x}}(R \bowtie S) \\ 0 & \text{otherwise.} \end{cases}$$

Classical Yannakakis. In its standard formulation, Yannakakis takes as input a *traditional* join tree T for a join query Q and a database db on Q. The algorithm starts by assigning a GMR R_n over var(n) to each node n in T. Initially, $R_n := db_n$. The algorithm then works in three stages.

(1) The nodes of T are visited in some bottom-up traversal order of T. When node n is visited in this order, its parent p is considered and R_p is updated to $R_p := R_p \ltimes R_n$.
(2) The nodes of T are visited in a top-down traversal order. When node n is visited in this order, each child c of n is considered, and R_c is updated to $R_c := R_c \ltimes R_n$.
(3) The interior nodes of T are again visited in a bottom-up order. In this stage, however, the actual join results are computed: when node n with children $c_1, \ldots, c_k$ is visited, its GMR is updated to $R_n := R_{c_1} \bowtie \cdots \bowtie R_{c_k}$.

After the final stage, the GMR materialized at the root is precisely $Q(db)$. The initialization together with stages (1) and (2) run in time $O(\|db\|)$ while stage 3 can be shown to

run in time $O(\|Q(db)\|)$. It is worth noting that the Yannakakis algorithm fully materializes the query result $Q(db)$ at the root, requiring $O(\|Q(db)\|)$ space. Notice also that this algorithm works over the static setting, and does not consider updates.

To extend Yannakakis to work on generalized join trees in addition to traditional join trees, one only needs to modify the initialization step as follows. If n is a hyperedge, simply set $R_n := \pi_{\text{var}(n)}\, R_c$ for some arbitrary but fixed $c \in \text{grd}(n)$ (which we may assume to have been initialized before if we initialize in a bottom-up fashion). It is not difficult to see that, with this initalization, every hyperedge n has $R_n = \pi_{\text{var}(n)}\, db_{\boldsymbol{a}}$ for some descendant atom $\boldsymbol{a}$ of n. In other words, R_n is the projection of some input atom. This ensures that, even on generalized join trees, Yannakakis exhibits the same complexity guarantees.

Yannakakis with constant delay enumeration (CDY). Our dynamic query processing algorithm is based on the simple observation that, after the first bottom-up traversal stage, the join query result $Q(db)$ can be enumerated with constant delay. As such, there is no need to materialize the query result in stage 3. To illustrate this claim, consider the following variant of the Classical Yannakakis algorithm, called CDY for Constant Delay Yannakakis.

(1) Do the first stage of Classical Yannakakis.
(2) For each node n construct an index L_n of R_n on pvar(n).

Given this pre-processing, the constant-delay enumeration method ENUM is essentially a multi-way hash join, where the GMR materialized at the root is used as probe relation, and the other R_n as build relations, with the hash tables given by L_n. Because of the way in which R_n is computed, we are ensured that for every probe we will have matching join tuples, ensuring constant-delay enumeration. Note, moreover, that the GMRs materialized after the first step of the Yannakakis algorithm, as well as the constructed indexes, require $O(\|db\|)$ space. We delay a formal definition of the enumeration algorithm until Section 4.5, but illustrate its working by means of the following example.

Example 4.3. Consider generalized join tree T_3 of Figure 2. ENUM works as follows. Let $\vec{s}$ be the empty tuple. Then ENUM is defined by:

for each $\vec{t}_{[y]} \in L_{[y]}(\vec{s})$ **do**
  for each $\vec{t}_{[x,y,z]} \in L_{[x,y,z]}(\vec{t}_{[y]})$ **do**
    for each $(\vec{t}_R, \mu_R) \in L_{R(x,y,z)}(\vec{t}_{[x,y,z]})$ **do**
      for each $(\vec{t}_S, \mu_S) \in L_{S(y,v,w)}(\vec{t}_{[y,v,w]}[y,v])$ **do**
        for each $\vec{t}_{[y,v]} \in L_{[y,v]}(\vec{t}_{[y]})$ **do**
          for each $(\vec{t}_T, \mu_T) \in L_{T[y,v,w]}(\vec{t}_{[y,v]})$ **do**
            for each $(\vec{t}_U, \mu_T) \in L_{U[y,v,p]}(\vec{t}_{[y,v]})$ **do**
              output $(\vec{t_R t_S t_T t_U}, \mu_R * \mu_S * \mu_T * \mu_U)$ □

From our discussion so far, we obain:

Proposition 4.4. *Given an acyclic full join query Q, a join tree T of Q and a database db, CDY(T, db) runs in time $O(\|db\|)$ using space $O(\|db\|)$. Once CDY has completed,* ENUM *effectively enumerates $Q(db)$ with constant delay.*

4.3 Dynamic Yannakakis

Definition 4.5. Let T be a join tree and db a database. Let R_n, for $n \in T$, be the GMR associated to n after executing the first stage of the Yannakakis algorithm. A tuple $\vec{t}$ is called *live* in (db, n) w.r.t. T if $\vec{t} \in R_n$.[2]

CDY shows that we can suitably index the live tuples to enumerate $Q(db)$ with constant delay. To turn CDY into a dynamic algorithm, it hence suffices to maintain the live tuples and indices under updates. A naive approach for doing this would be to re-run CDY from scratch whenever the database is updated. This would spend time linear in the size of the updated database. Of course, this naive approach introduces unnecessary overhead. Indeed, consider an update that inserts a single tuple to an atom $\boldsymbol{a}$. In that case, only the set of live tuples associated to $\boldsymbol{a}$ and its ancestors in join tree T can change, while the rest of the nodes would remain unchanged. Moreover, the new set of live tuples of $\boldsymbol{a}$ and its ancestors can be computed incrementally. In At the end of Section 5, we will see in particular that avoiding naive recomputation is highly effective in practice.

In order to be able to explain how we maintain the live tuples incrementally, we require the following definitions.

Definition 4.6. Let T be a join tree. To every node n of T we associate two queries, Λ_n^T and Ψ_n^T, over var(n) and pvar(n), respectively. To every hyperedge n of T we also associate an additional query Γ_n^T over var(n). The definition of these queries is recursive: for each atom $\boldsymbol{a}$ we define $\Lambda_{\boldsymbol{a}}^T$ simply as $\boldsymbol{a}$, and $\Psi_{\boldsymbol{a}}^T := \pi_{\text{pvar}(\boldsymbol{a})}\, \boldsymbol{a}$. Then, in a bottom-up traversal order, for every hyperedge n we define

$$\Lambda_n^T := \Gamma_n^T \bowtie \bowtie_{c \in \text{ng}(n)} \Psi_c^T \qquad \Psi_n^T := \pi_{\text{pvar}(n)}\, \Lambda_n^T$$
$$\Gamma_n^T := \bowtie_{c \in \text{grd}(n)} \Psi_c$$

We often omit the superscript if it is clear from the context. Intuitively, Λ_n contains the set of live tuples of n, while Ψ_n and Γ_n are auxiliary queries that will help maintain Λ_n under updates. The following proposition shows that, indeed, the queries Λ_n characterize the live tuples. The proof (omitted) is by induction on the height of node n in T.

Proposition 4.7. *A tuple $\vec{t}$ is live in (db, n) w.r.t. join tree T if, and only if, $\vec{t} \in \Lambda_n^T(db)$.*

We can now define the data structure maintained by DYN.

Definition 4.8 (T-representation)**.** Let T be a join tree and let db be a database. A *T-representation* (T-rep for short) of db is a data structure $\mathcal{D}$ that for each node n of T contains:
- an index L_n of $\Lambda_n(db)$ on pvar(n);
- a GMR P_n that materializes $\Psi_n(db)$, i.e., $P_n = \Psi_n(db)$;
- a GMR G_n that materializes $\Gamma_n(db)$, i.e., $G_n = \Gamma_n(db)$;
- for every non-guard child $c \in \text{ng}(n)$, an index $G_{n,c}$ of G_n on pvar(c).

Example 4.9. Consider join tree T_3 from Figure 2. In Figure 3, we show the T_3-rep $\mathcal{D}$ for the database db consisting of the GMRs $R(x,y,z)$, $S(x,y,u)$, $T(y,v,w)$, $U(y,v,p)$ presented at the leaves of Figure 3. For each node n, the live tuples $L_n = \Lambda_n(db)$ are given by the white-colored tables (shown below n) while $P_n = \Psi_n(db)$ is given by the gray-colored tables (shown above n on the edge from n to its parent). For reasons of parsimony, we do not show the G_n: for $[y]$ and $[y,v]$ this equals L_n; for $[x,y,z]$ this equals $L_{R(x,y,z)}$. The indexes are likewise not shown. □

A first important feature of T-representations is that they use only linear space.

[2]Recall that $\vec{t} \in S$ indicates that $S(\vec{t}) \neq 0$.

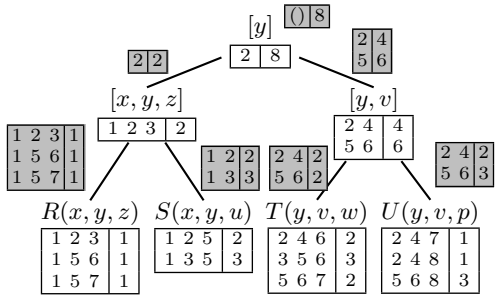


Figure 3: Illustration of T-representations (Example 4.9).

Proposition 4.10. *Let T be a join tree and $\mathcal{D}$ a T-rep of db. Then $\|\mathcal{D}\| = O(\|db\|)$.*

The crux to prove this proposition lies in observing that, as illustrated in Figure 3, for all nodes n, if $\vec{t}$ is in $\Lambda_n(db)$, $\Psi_n(db)$, or $\Gamma_n(db)$, then there is some descendant atom $\boldsymbol{a} \in T$ such that $\vec{t} \in \pi_{\overline{x}}(db_{\boldsymbol{a}})$ with $\overline{x} = \mathrm{var}(n)$ or $\overline{x} = \mathrm{pvar}(n)$. Therefore, $\Lambda_n(db)$, $\Psi_n(db)$, and $\Gamma_n(db)$ as well as indexes thereon, all take space $O(\|db\|)$.

Dynamic Yannakakis. We now describe the Dynamic Yannakakis algorithm (Dyn) presented in Algorithm 1. Dyn maintains T-representations under updates. To explicitly assert the join-tree over which Dyn operates we write $\textsc{Dyn}_T$.

Like classical Yannakakis, $\textsc{Dyn}_T$ traverses the nodes of T in a bottom-up fashion upon update u. During this traversal, the goal is to materialize, for each node n, the deltas $\Delta\Lambda_n(db, u)$, $\Delta\Psi_n(db, u)$, and $\Delta\Gamma_n(db, u)$ into GMRs ΔL_n, ΔP_n, and ΔG_n, respectively. These represent the updates that we need to apply to $\mathcal{D}$'s components in order to obtain a T-rep for $db + u$. This application happens in lines 6–10.

The delta GMRs are computed as follows. When n is an atom $\boldsymbol{a}$, $\textsc{Dyn}_T$ uses the update u to compute $\Delta L_{\boldsymbol{a}} = u_{\boldsymbol{a}}$ and $\Delta P_{\boldsymbol{a}} = \pi_{\mathrm{pvar}(\boldsymbol{a})}\, u_{\boldsymbol{a}}$. The latter projection can be done using a simple hash-based aggregation algorithm. When n is a hyperedge, $\textsc{Dyn}_T$ uses Algorithm 2 to compute ΔL_n, ΔP_n and ΔG_n. This algorithm uses the materialized index of $\Lambda_n(db)$ on $\mathrm{pvar}(n)$, and the materialized GMRs $P_n = \Psi_n(db)$ and $G_n = \Gamma_n(db)$, which are already available in $\mathcal{D}$. In addition, it uses the delta GMRs ΔP_c for each child c of n, which was previously computed when visiting c. In order to compute ΔL_n, ΔP_n, and ΔG_n efficiently, we use the following insight, proven in the Appendix B.

Lemma 4.11. *If $\vec{t} \in \Delta\Gamma_n(db, u)$ then $\vec{t} \in \Delta\Psi_c(db, u)$ for some guard $c \in \mathrm{grd}(n)$. Moreover, if $\vec{t} \in \Delta\Lambda_n(db, u)$ then either (1) $\vec{t} \in \Delta\Psi_c(db, u)$ for some guard $c \in \mathrm{grd}(n)$ or (2) $\vec{t} \in (\Gamma_n(db) \ltimes \Delta\Psi_c(db, u))$ for some child $c \in \mathrm{ng}(n)$.*

Algorithm 2 uses Lemma 4.11 to compute a bound on $\mathrm{supp}\,(\Delta\Lambda_n(db, u))$. In particular, in lines 2–4 it computes

$$U = \bigcup_{c \in \mathrm{grd}(n)} \mathrm{supp}(\Delta P_c) \cup \bigcup_{c \in \mathrm{ng}(n)} \mathrm{supp}(G_n \ltimes \Delta P_c).$$

As such, U contains all tuples that can appear in $\Delta\Gamma_n(db, u)$ or $\Delta\Lambda_n(db, u)$. Lines 5–8 compute $\Delta G_n = \Delta\Gamma_n(db, u)$, $\Delta L_n = \Delta\Lambda_n(db, u)$ and $\Delta P_n = \Delta\Psi_n(db, u)$ by iterating over the tuples in U and using the fact that $\Psi_c(db + u)(\vec{s}) = P_c(\vec{s}) + \Delta P_c(\vec{s})$, for every tuple $\vec{s}$. From Lemma 4.11 and our explanation so far, we hence obtain:

Algorithm 1 $\textsc{Dyn}_T$: Update trigger maintaining T-rep $\mathcal{D}$ under update u

```
1: Assume: T is a join tree
2: Input: T-rep D for (db); and update u
3: Output: T-rep for db + u.
4: for each node n ∈ T, visited in bottom-up order do
5:   compute ΔL_n, ΔP_n, and ΔG_n (if applicable)
6: for each node n ∈ T do
7:   L_n += ΔL_n ;      P_n += ΔP_n
8:   if n is a hyperedge then
9:     G_n += ΔG_n
10:    for each c ∈ ng(n) do G_{n,c} += ΔG_n
```

Algorithm 2 Delta computation for hyperedge n

```
1: Initialize ΔL_n, ΔP_n and ΔG_n to the empty GMRs
2: Initialize U := ⋃_{c∈grd(n)} supp(ΔP_c)
3: for each c ∈ ng(n) and each t_c ∈ ΔP_c do
4:   U := U ∪ supp(G_{n,c}[t_c])
5: for each t ∈ U do
6:   ΔG_n[t] := ∏_{c∈grd(n)} (P_c + ΔP_c)[t] − G_n[t]
7:   ΔL_n[t] := ∏_{c∈ch(n)} (P_c + ΔP_c)[t[pvar(c)]] − L_n[t]
8:   ΔP_n[t[pvar(n)]] += ΔL_n[t]
```

Theorem 4.12. *Let $\mathcal{D}$ be a T-rep for db and let u be an update to db. $\textsc{Dyn}_T(\mathcal{D}, u)$ produces a T-rep for $db + u$.*

4.4 Complexity of Dynamic Yannakakis

Next, we study the efficiency with which $\textsc{Dyn}_T$ maintains T-reps under updates. Towards this end, we first illustrate $\textsc{Dyn}_T$'s operation by example.

Example 4.13. Consider join tree T_3 from Figure 2 and the T_3-rep $\mathcal{D}$ shown in Figure 3 that we discussed in Example 4.9. Consider the update $\{(1, 5, 9) \mapsto 2\}$ on S (i.e., tuple $(1, 5, 9)$ is inserted into $S(x, y, u)$ with multiplicity 2). When $\textsc{Dyn}_{T_3}$ executes, it will compute empty delta GMRs for all nodes, except for $S(x, y, u)$ and its ancestors. In particular, when Algorithm 2 is run on hyperedge $[x, y, z]$, it operates as follows. First observe that $[x, y, z]$ has only one guard child, namely $R(x, y, z)$. Hence $G_{[x,y,z]} = P_{R(x,y,z)} = L_{R(x,y,z)} = R$. Since R is not updated, U is initialized to $\emptyset$ in line 2. Then, in lines 3 and 4, Algorithm 2 uses the index $G_{[x,y,z],S[x,y,u]}$ of $G_{[x,y,z]}$ on $[x, y]$ to directly retrieve all tuples in $R(x, y, z)$ that satisfy $x = 1$ and $y = 5$. This is the main purpose of the indexes $G_{n,c}$: we do not have to iterate over the entire GMR G_n nor check for equalities. During the rest of its computation, Algorithm 2 then calculates $\Delta G_{[x,y,z]} = \emptyset$, $\Delta L_{[x,y,z]} = \{(1, 5, 6) \mapsto 2, (1, 5, 7) \mapsto 2\}$, and $\Delta P_{[x,y,z]} = \{(5) \mapsto 4\}$. When processing $[y]$, we have to *propagate* $\Delta P_{[x,y,z]}$ from $[x, y, z]$ to $[y]$. This is done by initializing $U = \{(5)\}$ in line 2 of Algorithm 2, after which $\Delta G_{[y]} = \Delta L_{[y]} = \{(5) \mapsto 24\}$ and $\Delta P_{[y]} = \{() \mapsto 24\}$. □

It is important to observe that in this example the single-tuple update $\{(1, 5, 9) \mapsto 2\}$ on S triggers *multiple* tuples to become live in $[x, y, z]$. This occurs simply because the variable z of $[x, y, z]$ is not in $S(x, y, u)$; therefore a single-tuple update to S can match many tuples in $G_{[x,y,z]}$ with different z values. In fact, in the worst case, it may cause

as many tuples to become live in $[x, y, z]$ as there are tuples in $G_{[x,y,z]} = R$. In contrast, single-tuple updates into $R(x, y, z), T(y, v, w)$, or $U(y, v, p)$, can cause at most 1 tuple to become live in any of their parents. This is because R's (resp. T's and U's) parent contains only variables that are also mentioned in R (resp. T, resp. U). Likewise, updates to $[x, y, z]$ (resp. $[y, v]$) that we need to propagate to $[y]$ can only cause as many $[y]$ tuples to become live as have become live in $[x, y, z]$ (resp. $[y, v]$).

So, the fact that a node contains all variables mentioned in its parent makes it efficient to propagate updates from that node to its parent. Trees for which all nodes (except for the root) contain all variables mentioned in their parent are called simple trees.

Definition 4.14 (Simplicity)**.** A width-one GHD T is *simple* if every child node in T is a guard of its parent. A query Q is simple if it has a simple join tree.

For example, T_4 of Figure 2 is simple, but T_3 is not since $S(x, y, u)$ is a child but not a guard of $[x, y, z]$.

Because in simple trees the number of tuples that can propagate from a child update to its parent is bounded by the size of the child update, we obtain that, for a simple tree T, DYN_T maintains a T-rep under update u in time linear in u, independent of the databse *db*.

Theorem 4.15. $\text{DYN}_T(\mathcal{D}, u)$ *produces a T-rep for $db + u$ in time $O(\|u\|)$ for every database db and every update u if, and only if, T is simple.*

The technical proof is deferred to the full version of this paper because of space constraints.

Theorem 4.15 indicates that, before using DYN to dynamically process Q, it is important to check for the existence of a generalized simple join tree for Q.

On non-simple trees T, such as the tree T_3 from Example 4.13, DYN_T is less efficient in the worst case. Indeed, as already illustrated above, a single-tuple update can trigger multiple-tuple updates to its ancestors and in the worst case the parent update may be as big as $\|db\|$. In principle, the multiple-tuple update to the parent may cause an even bigger update to the grand-parent (assuming that the latter is not a guard of the grand-parent). The number of tuples in an update to a node can be shown to be always bounded by $\|db\| + \|u\|$, however. Using this observation, we can show:

Proposition 4.16. *Let T be a join tree, $\mathcal{D}$ a T-rep for db and u an update to db.* $\text{DYN}_T(\mathcal{D}, u)$ *produces a T-rep for $db + u$ in time $O(\|db\| + \|u\|)$.*

In other words, in the worst case, DYN_T runs in time $O(\|db\| + \|u\|)$, which is unfortunately similar to recomputing everything from scratch using CDY after every update. Fortunately, while recomputing everything from scratch will always cost $\Omega(\| db + u \|)$ time, in practice DYN performs much better than its $O(\|db\| + \|u\|)$ upper bound. This is discussed at the end of Section 5.

4.5 Enumeration

In this section, we show that a T-rep $\mathcal{D}$ for *db*, with T a join tree for join query Q can be used not only to enumerate $Q(db)$ with constant delay, but also some of its projections $\pi_{\overline{x}}Q(db)$. In particular, CDE of projections is possible if there exists a subtree of T that includes the root and contains precisely the set $\overline{x}$ of projected variables. Intuitively, if such subtree exists, the enumeration algorithm will be able to *traverse* $\mathcal{D}$ to find the required values of $\overline{x}$ without traversing tuples containing variables in $\text{var}(Q)\backslash\overline{x}$, which may cause unbounded delays in the enumeration. Essentially the same idea has been used before in the context of static CDE [5] (as discussed in Section 4.6), factorized databases [6], and worst-case optimal algorithms [23]. We proceed formally.

Algorithm 3 $\text{ENUM}_{(T,N)}(\mathcal{D})$

```
for each (t⃗_{n_1}, μ_{n_1}) ∈ L_{n_1}([]) do
  for each (t⃗_{n_2}, μ_{n_2}) ∈ L_{n_2}(t⃗_{p(n_2)}[x̄_{n_2}]) do
    for each (t⃗_{n_3}, μ_{n_3}) ∈ L_{n_3}(t⃗_{p(n_3)}[x̄_{n_3}]) do
      ...
      for each (t⃗_{n_k}, μ_{n_k}) ∈ L_{n_k}(t⃗_{p(n_k)}[x̄_{n_k}]) do
        let μ = μ_{c_1} * ··· * μ_{c_l} * Π_{m∈M} P_m(t⃗_{p(m)}[x̄_m])
        output (t⃗_{c_1} t⃗_{c_2} ... t⃗_{c_l}, μ)
```

Definition 4.17. Let $T = (V, E)$ be a join tree. A subset $N \subseteq V$ is *connex* if it includes the root and the subgraph of T induced by N is a tree.

To illustrate, $\{[y], [x, y, z], [y, v]\}$ is a connex subset of the join tree T_3 of Figure 2, but $\{[y], S(x, y, u)\}$ is not.

For each join tree T and each connex subset N of its nodes we define enumeration algorithm $\text{ENUM}_{(T,N)}$ as follows. Let T' be the subtree of T induced by N and let $(n_k, n_{k-1}, \ldots, n_1)$ be a topological sort of T'. In particular, n_1 is the root of T. Let $c_1, \ldots, c_l$ be the leaf nodes of T'. Let $p(n)$ denote the parent of node n and let $\overline{x_n}$ denote $\text{pvar}(n)$. Finally, let M be the subset of all nodes in T that are not in N, but for which some sibling is in N. With this notation, $\text{ENUM}_{(T,N)}$ is shown in Algorithm 3. Example 4.3 shows $\text{ENUM}_{(T_3,N)}(\mathcal{D})$ for the join tree T_3 of Figure 2 with N consisting of all nodes in T_3.

Proposition 4.18. *Let T be a join tree for join query Q. Assume that N is a connex subset of T. Then* $\text{ENUM}_{(T,N)}(\mathcal{D})$ *enumerates $Q'(db) := \pi_{\text{var}(N)}Q(db)$ with constant delay, for every database db and every T-rep $\mathcal{D}$ of db. Moreover, for an arbitrary tuple $\vec{t}$, its multiplicity in $Q'(db)$ can be calculated from $\mathcal{D}$ in constant time.*

The technical proof is deferred until the full version of this paper. Note that $\text{ENUM}_{T,V}(\mathcal{D})$ with V the set of all nodes in T enumerates $Q(db)$. Combining all of our results so far we obtain that join-tree representations are DCLRs for the class of all *free-connex acyclic CQs*, which is defined as follows.

Definition 4.19. (Compatible, Free-Connex Acyclic) Let T be a join tree. A CQ Q is *compatible with* T if T is a join tree for Q and T has a connex subset N with $\text{var}(N) = out(Q)$. A CQ is *free-connex acyclic* if it has a compatible join tree.

In particular, every acyclic join query is free-connex acyclic. Let T be a join tree. It now follows that the class of all T-reps form a DCLR of every CQ Q compatible with T.

Delta-enumeration. Using DYN_T we can actually also enumerate deltas $\Delta Q(db, u)$ of $Q(db)$ under single-tuple update u. This result is relevant for *push-based* query processing systems, where users do not ping the system for the complete current query answer, but instead ask to be notified of the changes to the query results when the database changes. In addition, as we will discuss in Section 5, it also provides a key method for dynamic processing of CAQs.

Definition 4.20 (ΔT-rep)**.** Let T be a join tree, db be a database and let u be an update to db. A *ΔT-representation* of (db, u) is a data structure $\Delta\mathcal{D}$ that contains (1) a T-rep for db; (2) an index ΔL_n of $\Delta\Lambda_n(db, u)$ on pvar(n), for every $n \in T$; and (3) a GMR ΔP_n that materializes $\Delta\Psi_n(db, u)$, for every $n \in T$.

Note that DYN$_T$ needs to compute $\Delta L_n = \Delta\Lambda_n(db, u)$ and $\Delta P_n = \Delta\Psi_n(db, u)$ anyway when processing T-rep $\mathcal{D}$ under update u. Hence, after the bottom-up pass in lines 4–5 of DYN$_T$ we obtain a ΔT-rep $\Delta\mathcal{D}$, provided that we represent ΔL_n as an index on pvar(n).

Theorem 4.21. *Let u be a single-tuple update to database db. Let $\Delta\mathcal{D}$ be a ΔT-rep of (db, u) and let Q be compatible with T. Then $\Delta Q(db, u)$ can be enumerated with constant delay from $\Delta\mathcal{D}$.*

Due to space constraints, we defer the definition of the delta-enumeration algorithm to the full version of this paper. How to enumerate $\Delta Q(db, u)$ with constant delay for general, multiple-tuple updates remains an open problem.

4.6 Optimalitiy

In this section we show that Dynamic Yannakakis is optimal in two aspects. (1) It is able to dynamically process the largest subclass of CQs for which DCLRs can reasonably be expected to exist. (2) The class of queries for which DYN processes updates in $O(\| u \|)$ time (Theorem 4.15) is the largest class of queries for which we can reasonably expect to have such update processing time as well as CDE of results.

DCLR-optimality. In the static setting without updates, a query Q is said to be in class CD $\circ$ LIN if there exists an algorithm that, for each database db does an $O(\| db \|)$-time precomputation and then proceeds in CDE of $Q(db)$, evaluated under set semantics. Bagan et al. showed that, under the so-called *binary matrix multiplication conjecture*, an acyclic CQ is in CD $\circ$ LIN if and only if it is free-connex [5]. Recently, Brault-Baron extended this result under two assumptions: the triangle hypothesis (checking the presence of a triangle in a hypergraph with n nodes cannot be done in time $O(n^2)$) and the tetrahedron hypothesis (for each $k > 2$, checking whether a hypergraph contains a k-simplex cannot be done in time $O(n)$). Under these assumptions, he shows that a CQ is in CD $\circ$ LIN if and only if it is free-connex acyclic [10]. In Appendix C we prove that this implies:

Proposition 4.22. *Under the above-mentioned hypotheses, a DCLR exists for CQ Q if, and only if, Q is free-connex acyclic.*

Processing-time optimality. Berkholz et al. have recently characterized the class of self-join free CQs that feature a representation that allows both CDE of results and $O(1)$ maintenance under single-tuple updates [7]. In particular, they show that, under the assumption of hardness of the Online Matrix-Vector Multiplication problem [22], the following dichotomy holds. When evaluated under set semantics, a CQ Q without self joins features a representation that supports CDE and maintenance in $O(1)$ time under single-tuple updates if, and only if, Q is *q-hierarchical*. Notice Q can be maintained in $O(1)$ time under single-tuple updates if, and only if, it can also be maintained in $O(\|u\|)$ time under arbitrary updates. The definition of *q-hierarchical* queries is the following.

Definition 4.23 (q-hierarchicality)**.** Given a CQ Q and a variable $x \in \text{var}(Q)$, let $at(x)$ denote the set of all atoms in which x occurs in Q. Q is called *hierarchical* if for every pair of variables $x, y \in \text{var}(Q)$, either $at(x) \subseteq at(y)$ or $at(y) \subseteq at(x)$ or $at(x) \cap at(y) = \emptyset$. A CQ Q is q-hierarchical if it is hierarchical and for every two variables $x, y \in \text{var}(Q)$, if $x \in out(Q)$ and $at(x) \subsetneq at(y)$, then $y \in out(Q)$.

Example 4.24. Consider the join query $Q = R(x, y, z) \bowtie S(x, y, u) \bowtie T(y, v, w) \bowtie U(y, v, p)$. Q is hierarchical. Moreover, $\pi_{x,y}Q$ is q-hierarchical. In contrast, $\pi_u Q$ is not q-hierarchical since $u \in out(Q)$, $at(u) \subsetneq at(y)$, yet $y \notin out(Q)$.

Observe that a join query is q-hierarchical iff it is hierarchical. The hierarchical property has actually already played a central role for efficient query evaluation in various contexts [16, 17, 27], see [7] for a discussion.

The following two propositions (proven in Appendix C) establish the relationship between DYN and the dichotomy of Berkholz et al.

Proposition 4.25. *If a CQ Q is q-hierarchical, then it has a join tree which is both simple and compatible.*

It then follows from Theorem 4.15 and Proposition 4.18 that, for q-hierarchical queries, DYN also processes single-tuple updates in $O(1)$ time while allowing the query result to be enumerated with constant delay (given the join tree). This hence matches the algorithm provided by Berkholz et al. for processing q-hierarchical queries under updates. Note that, by Proposition 4.25, all q-hierarchical queries must be free-connex acyclic.

Proposition 4.26. *If a CQ Q has a join tree T which is both simple and compatible with Q, then Q is q-hierarchical.*

This result is to be expected, since from Theorem 4.15 and Proposition 4.18 we know that for such T we can do CDE of Q and do maintenance under updates in $O(\|u\|)$ time. If other than q-hierarhical queries had simple compatible join trees, Berkholz et al.'s dichotomy would fail. Also observe that, as seen in Section 4.4, DYN may process updates in $\omega(\| u \|)$ time on non-simple join trees. Berkholz et al.'s dichotomy implies that this is unavoidable in the worst case.

At this point, we can explain why it is important to work with join trees based on width-one GHDs rather than classical join trees (which do not allow partial hyperedges to occur). Indeed, the following proposition (proven in Appendix C) shows that there are hierarchical queries for which no classical join tree is simple. Therefore, if we restrict ourselves to classical join trees we will fail to obtain an $O(\|u\|)$ update time for some q-hierarchical queries.

Proposition 4.27. *Let Q be the hierarchical join query $R(x, y, z) \bowtie S(x, y, u) \bowtie T(y, v, w) \bowtie U(y, v, p)$. Every simple width-one GHD for Q has at least one partial hyperedge.*

5. IMPLEMENTATION AND EXPERIMENTAL VALIDATION

In this section, we experimentally measure the performance of DYN, focusing on both throughput and memory consumption. We start by describing how our implementation addresses some practical issues, then we describe in detail the operational setup, and finally present the experimental results.

5.1 Practical Implementation

We have described how DYN processes free-connex acyclic CQs under updates. In this subsection, we first explain how to use DYN as an algorithmic core for practical dynamic query evaluation of the more general class of acyclic conjunctive aggregate queries (not necessarily free-connex).

Definition 5.1. A conjunctive aggregate query (CAQ) is a query of the form $Q' = (\overline{x}, f_1, \ldots, f_n)Q$, where Q is a CQ; $\overline{x} \subseteq out(Q)$ and f_i is an aggregate function over $out(Q)$ for $1 \leq i \leq n$. Q' is *acyclic* if its CQ Q is so.

Example aggregate functions are $\text{SUM}(u) \times 3$ or $\text{AVG}(x)$, assuming u, x in $out(Q)$. The semantics of CAQs is best illustrated by example. Consider $Q' = (x, y, \text{AVG}(v))\,\pi_{x,y,v}$ $(R(x,y,z) \bowtie S(y,z,v))$. This query groups the result of $\pi_{x,y,v}(R(x,y,z) \bowtie S(y,z,v))$ by x, y, and computes $\text{AVG}(v)$ (under multiset semantics) for each group. It should be noted that we assume that the aggregate functions need to be *streamable*. This means that one should be able to update the aggregate function results by only inspecting the updates to $Q(db)$ and the previous aggregate value plus, possibly, a constant amount of extra information per tuple

We can dynamically process an acyclic CAQ Q' using DYN by means of a simple strategy: use DYN to maintain a DCLR for the acyclic CQ Q of Q', but materialize the output of the CAQ in an array. Use delta-enumeration on Q to maintain this array under single-tuple updates. Note that, in order to support delta-enumeration with constant delay, we require that Q is free-connex (Theorem 4.21). If this is not the case, (which frequently occurs in practice), we let DYN maintain a DCLR for 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 by adding all variables to the output). Of course, under this strategy, we require $\Omega(\|Q'(db)\|)$ space, just like (H)IVM, but we avoid the (partial) materialization of Q and its deltas. As shown in Section 5.3, this property actually make DYN outperform HIVM in both processing time and space.

An important optimization that our implementation applies in this context, is that of early computation of aggregate functions that are restricted to variables of a single atom. For example, consider $Q' = (x, y, \text{SUM}(t))\,\pi_{x,y,t}$ $(R(x,y,t) \bowtie S(y,z,v))$. Our implementation will actually run DYN on $\pi_{x,y}(R'(x,y) \bowtie S(y,z,v))$ where R' is the GMR that maps tuple $(x,y) \mapsto \sum_t t \times R(x,y,t)$. Note that R' can be maintained under updates to R.

Sub-queries Before proceeding to the experimental evaluation of DYN, we briefly discuss how to evaluate queries with sub-queries. Recall from Proposition 4.18 that T-reps have a particularly interesting property: If $\mathcal{D}$ is a T-rep and Q is compatible with T, then the multiplicity of an arbitrary tuple $\vec{t}$ in $Q(db)$ can be calculated in constant time from $\mathcal{D}$. This is highly relevant in practice, since when evaluating queries with IN or EXIST sub-queries, it suffices to maintain two DCLRs, one for the subquery and one for the outer query. From the viewpoint of the outer query, the subquery DCLR then behaves as an input GMR.

Generalized Join Trees When dynamically processing a CQ, the join tree under consideration can impact the performance of DYN. For example, one would expect that when processing a q-hiearchical query, DYN performs better using a simple tree than a non-simple tree. One could measure *how simple* a tree is by estimating the amount of single-tuple updates that will be processed in constant time by DYN. Although there are well-known algorithms for heuristic search of hypertree decompositions [18], their objective is to find low-width decompositions, and therefore are not well-suited for our setting. We have developed a simple cost model for generalized join trees and have used minimum-cost trees for experimentation. For the sake of space, the details of this cost model are left to the full version of the paper.

Benchmark		Query	# of tuples
TPC-H	Full joins	**FQ1**	2,833,827
		FQ2	2,617,163
		FQ3	2,820,494
		FQ4	2,270,494
	Aggregate queries	**Q1**	7,999,406
		Q3	10,199,406
		Q4	9,999,406
		Q6	7,999,406
		Q9	11,346,069
		Q12	9,999,406
		Q13	2,200,000
		Q16'	1,333,330
		Q18	10,199,406
TPC-DS	Full joins	**FQ5**	10,669,570
	Aggregate queries	**Q3**	11,638,073
		Q7	13,559,239
		Q19	11,987,115
		Q22	36,138,621

Table 1: Number of tuples in the stream file of each query

5.2 Experimental Setup

Queries and update streams. We evaluate the subset of queries available in the industry-standard benchmarks TPC-H and TPC-DS that can be evaluated by the methods described throughout this paper. In particular, we evaluate those queries involving only equijoins, whose FROM-WHERE caluses are acyclic. Queries are divided into acyclic full-join queries (called FQs) and acyclic aggregate queries. Acyclic full join queries are generated by taking the FROM clause of the corresponding queries on the benchmarks. It is important to mention that we omit the ORDER BY and LIMIT clauses, we replaced the left-outer join in query Q13 by an equijoin, and modified Q16 to remove an inequality. We discard those queries using the MIN and MAX aggregate functions as this is not supported by our current implementation. We report all the evaluated queries in Appendix D.

Our update streams consist of single-tuple insertions only and are generated as follows. We use the data-generating utilities of the benchmarks, namely *dbgen* for TPC-H and *dsdgen* for TPC-DS[3]. We used scale factor 0.5 and 2 for the FQs from TPC-H and TPC-DS, respectively, and scale factor 2 and 4 for the aggregate queries from TPC-H and TPC-DS, respectively. Notice that the data-generating tools create datasets for a fixed schema, while most queries do not use the complete set of relations. The update streams are generated by randomly selecting the tuples to be inserted from the relations that occur in each query. To use the same update streams for evaluating both DYN and HIVM, each stream is stored in a file. The number of tuples on each file is depicted in Table 1.

[3] *dbgen* and *dsgen* are available at http://www.tpc.org/

Comparison to HIVM. As discussed in the introduction, HIVM is an efficient method for dynamic query evaluation that highly improves processing time over IVM [26]. We compare our implementation against DBToaster [26], a state-of-the-art HIVM engine. DBToaster is particularly meticulous in that it materializes only useful views, and therefore it is an interesting implementation for comparison in both throughput and memory usage. Moreover, DBToaster has been extensively tested and proven to be more efficient than a commercial database management system, a commercial stream processing system and an IVM implementation [26]. DBToaster compiles SQL queries into trigger programs for different programming languages. We compare against those in Scala, the same programming language used in our implementation. It is important to mention that programs compiled by DBToaster use the so-called *akka actors*[4] to generate update tuples. During our experiments, we have found that this creates an unnecessary memory overhead by creating many temporary objects. For a fair comparison we have therefore removed these actors from DBToaster.

Operational setup. The experiments are performed on a machine running GNU/Linux with an Intel Core i7 processor running at 3.07 GHz. We use version 2.11.8 of the Scala programming language, version 1.8.0_101 of the Java Virtual Machine, and version 2.2 of the DBToaster compiler. Each query is evaluated 10 times against each of the two engines for measuring time, and two times for measuring memory; the presented results are the average measurements over those runs. Every time a query is evaluated, 16 GB of main memory are freshly allocated to the corresponding program. To measure memory usage we use the Java Virtual Machine (JVM) System calls. We measure the memory consumption every 1000 updates, and consider the maximum value. For a fair comparison, we call the garbage collector before each memory measurement. The time used by the garbage collector is not considered in the measurements of throughput.

5.3 Experimental results

Figure 4 depicts the resources used by DYN as a percentage of the resources used by DBToaster. For each query, we plot the percentage of memory used by DYN considering that 100% is to the memory used by DBToaster, and the same is done for processing time. This improves readability and normalizes the chart. To present the absolute values, on top of the bars corresponding to each query we write the memory and time used by DBToaster. Some executions of DBToaster failed because they required more than 16GB of main memory. In those cases, we report 16GB of memory and the time it took the execution to raise an exception. We mark such queries with an asterisk (*) in Figure 4. Note that DYN never runs out of memory, and times reported for DYN are the times required to process the entire update stream.

Full-join queries. For full join queries (FQ1-FQ5), Figure 4 shows that DYN outperforms DBToaster by close to one order of magnitude, both in memory consumption and processing time. The difference in memory consumption is expected, since the result of full-join queries can be polynomially larger than the input dataset, and DBToaster materializes these results. The difference in processing time, then, is a consequence of DYN's maintenance of T-reps rather than the results themselves. The average processing time for DBToaster over FQ1-FQ5 is 128.49 seconds, while for DYN it is 29.85 seconds. This includes FQ1, FQ3, FQ4 and FQ5, for which DBToaster reached the memory limit. Then, 128.49 seconds is only a lower bound for the average processing time of DBToaster over FQ1-FQ5. Regarding memory consumption, DBToaster requires in average 14.68 GB for FQ1-FQ5 (considering a limit of 16 GB), compared to the 2.74 GB required by DYN. Note that the query presenting the biggest difference, FQ4, is a q-hierarchical query (see Section 4.4).

Aggregate queries. For aggregate queries, Figure 4 shows that DYN can significantly improve the memory consumption of HIVM while improving processing time up to one order of magnitude for TPC-H Q13' and TPC-DS Q7.

For TPC-H queries Q1, Q3, and Q6, DYN equals DBToasters memory consumption. For these queries, the algorithms used by DYN and DBToaster are nearly identical which is why DYN and DBToaster require the same amount of memory. The difference in execution time for these queries is due to implementation specifics. For example we have detected that DBToaster parses tuple attributes before filtering particular attributes in the WHERE clause. Our implementation, in contrast, does lazy parsing, meaning that each attribute is parsed only when it is used. In particular, if a certain attribute fails its local condition, then subsequent attributes are not parsed.

The biggest difference in processing time is observed for TPC-H query Q13' and TPC-DS query Q7. Q13' has a sub-query that computes the amount of orders processed for each customer. It then counts the number of customers for which k orders were processed, for each k. To process this, DBToaster almost fully recomputes the sub-query each time a new update arrives, which basically yields a quadratic algorithm. In contrast, our implementation also uses DYN to maintain the sub-query as a T-rep, supporting, for this particular case, constant update time. For Q7, the aggressive materialization of delta queries causes DBToaster to maintain 88 different GMRs. In contrast, to maintain its T-rep, DYN only needs to store 5 GMRs and 5 indexes.

Scalability. To show that DYN performs in a consistent way against streams of different sizes, we report the processing time and memory consumption each time a 10% of the stream is processed (Figure 6). The results show that for all queries the memory and time increase linearly with the amount of tuples processed. We can see that DYN is constantly faster and scales more consistently. The same phenomena occur for memory consumption. Due to space constraints, we only report the measurements for the TPC-H queries FQ1, Q3 and Q18.

Enumeration of query results. We know from Section 4.1 that T-reps feature constant delay enumeration, but this theoretical notion hides a constant factor that could decrease performance in practice. To show that this is not the case, we have measured the time needed for enumerating and writing to secondary memory the results of FQ1 to FQ4 from their corresponding DCLRs. We use update streams of different sizes, and for comparison we measure the time needed to iterate over the materialized set of results (from an in-memory array) and write them to secondary memory. The results are depicted in Figure 5. Interestingly, for larger result sizes, enumerating from a T-rep was slightly more efficient than enumerating from an in-memory array. A possible explanation is illustrated by the following exam-

[4]http://doc.akka.io/docs/akka/snapshot/scala/actors.html

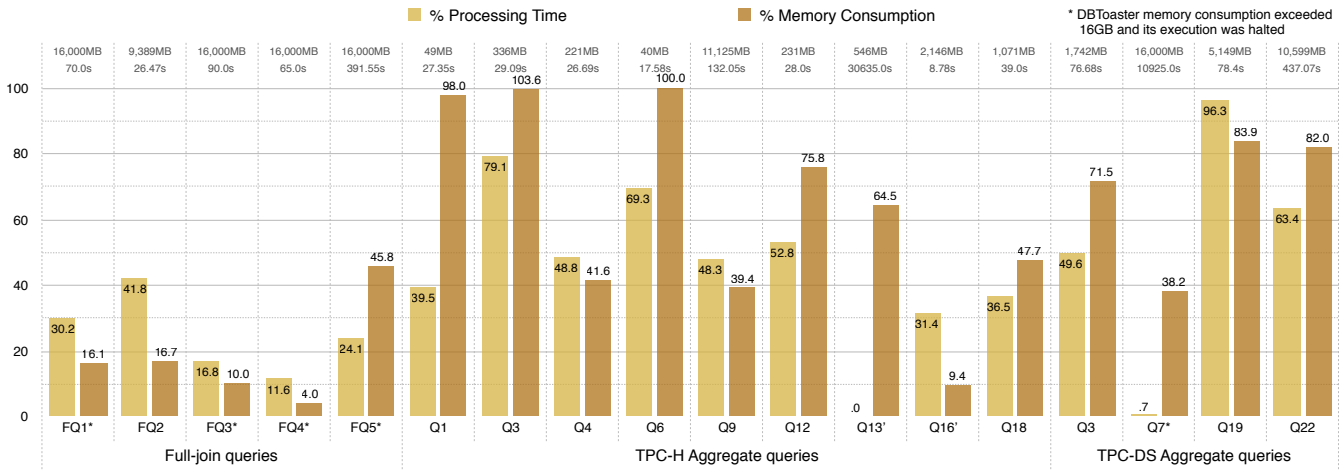


Figure 4: DYN usage of resources as a percentage of the resources consumed by DBToaster (lower is better).

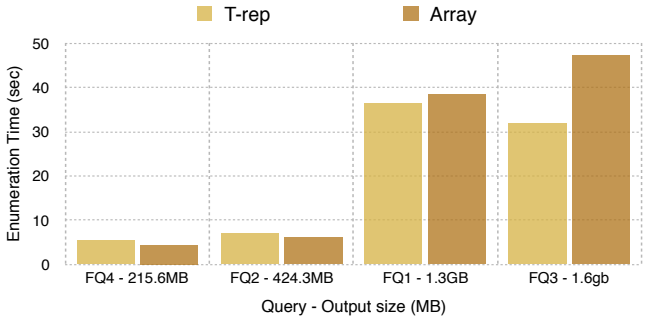


Figure 5: Time for enumerating output (lower is better)

ple. Consider the full-join query $R(A,B) \bowtie S(B,C)$, and assume there are several tuples in the join result. It is not hard to see that given a fixed B value, from a T-rep we can iterate over the C values corresponding to each A value. This way, the A and B values are not re-assigned while generating several tuples. In contrast, every time a tuple is read from an array each value needs to be read again.

Full query recomputation. In Section 4.3 we mentioned that, in theory, the worst-case complexity for updating a T-rep when T is not simple is the same as that of recomputing the Yannakakis algorithm from scratch. However, we can expect DYN to be much faster than the naive full-recomputation algorithm as it only updates those portions of the T-rep that are affected. This is indeed the case in practice. We tested both strategies over different datasets for FQ1 and FQ4. In average, the naive recomputation turned out to process updates 190 times slower than DYN. Due to space constraints we do not report the full results.

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.

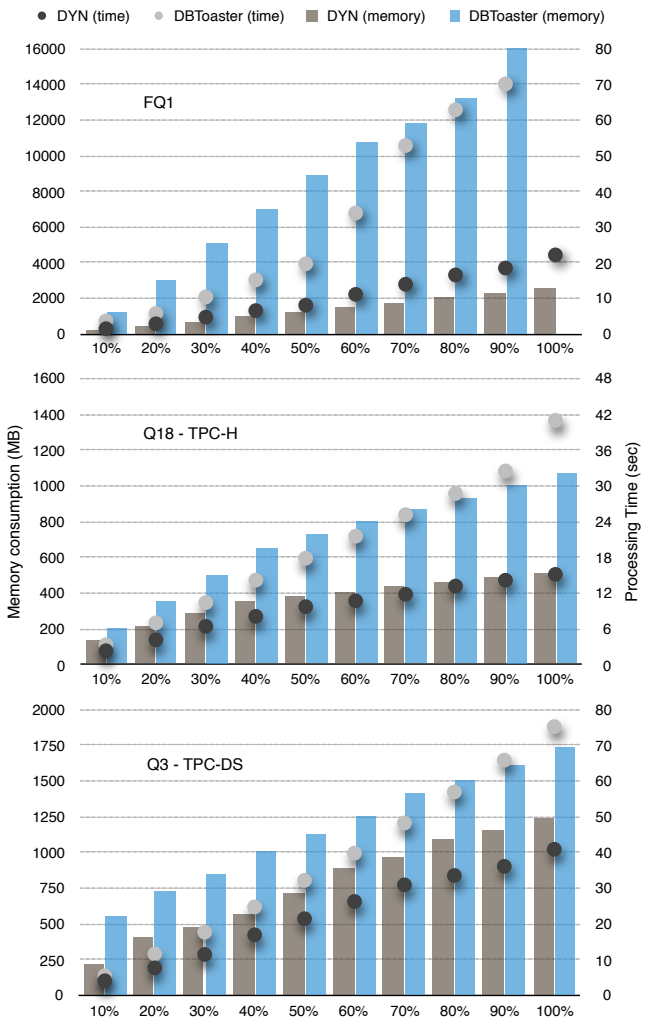


Figure 6: Resource utilization v/s % of tuples processed

6. REFERENCES

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

[2] M. Abo Khamis, H. Q. Ngo, and A. Rudra. FAQ: Questions asked frequently. In *Proc. of PODS*, pages 13–28, 2016.

[3] M. E. Adiba and B. G. Lindsay. Database snapshots. In *Proc. of VLDB 1980*, pages 86–91, 1980.

[4] S. M. Aji and R. J. McEliece. The generalized distributive law. *IEEE Trans. Information Theory*, 46(2):325–343, 2006.

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

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

[7] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering conjunctive queries under updates. In *Proc. of PODS*, 2017. To appear.

[8] J. A. Blakeley, N. Coburn, and P.-V. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM TODS*, (3):369–400, 1989.

[9] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. of SIGMOD*, pages 61–71, 1986.

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

[11] O. P. Buneman and E. K. Clemons. Efficiently monitoring relational databases. *ACM TODS*, (3):368–382, 1979.

[12] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of VLDB*, pages 577–589, 1991.

[13] R. Chirkova and J. Yang. *Materialized Views*. Now Publishers Inc., Hanover, MA, USA, 2012.

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

[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] N. Dalvi and D. Suciu. The dichotomy of probabilistic inference for unions of conjunctive queries. *J. ACM*, 59(6):30:1–30:87, 2013.

[17] R. Fink and D. Olteanu. Dichotomies for queries with negation in probabilistic databases. *ACM TODS*, 41(1):4:1–4:47, 2016.

[18] G. Gottlob, M. Grohe, n. Musliu, M. Samer, and F. Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In *Proc. of WG*, pages 1–15, 2005.

[19] G. Gottlob, N. Leone, and F. Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. *J. Comput. Syst. Sci.*, 66(4):775–808, 2003.

[20] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. of SIGMOD*, pages 157–166, 1993.

[21] H. Gupta and I. S. Mumick. Incremental maintenance of aggregate and outerjoin expressions. *Information Systems*, (6):435–464, 2006.

[22] 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.

[23] M. R. Joglekar, R. Puttagunta, and C. Ré. Ajar: Aggregations and joins over annotated relations. In *Proc. of PODS*, pages 91–106, 2016.

[24] A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. Implementing incremental view maintenance in nested data models. In *Proc. of DBPL*, pages 202–221, 1997.

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

[26] 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.

[27] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *Proc. of PODS*, pages 223–234, 2011.

[28] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. *SIGMOD Records*, 26(2):100–111, 1997.

[29] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD Records*, 42(4):5–16, 2014.

[30] 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.

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

[32] C. H. Papadimitriou. Computational complexity. In *Encyclopedia of Computer Science*, pages 260–265. 2003.

[33] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. on Knowl. and Data Eng.*, 3(3):337–341, 1991.

[34] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proc. of SIGMOD*, pages 447–458, 1996.

[35] N. Roussopoulos. Materialized views and data warehouses. *SIGMOD Records*, 27(1):21–26, 1998.

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

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

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

[39] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. of VLDB*, pages 82–94, 1981.

APPENDIX

A. PROOFS FROM SECTION 3

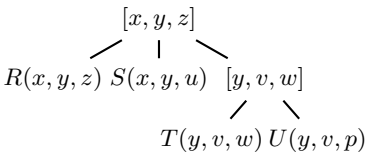
Readers familiar with GHDs of arbitrary width may observe that GHDs are normally defined as triples (T, χ, λ) with T a tree; χ a function that assigns a set of variables to each node, and λ a function that assigns a set of atoms to each node (see [19]). Since we focus on GHDs of width one, and hence do not need the full richness of GHDs, we omit χ and λ from our definition. These can be recovered by fixing $\chi\colon n \to \text{var}(n)$ and λ to be the function that maps atoms $\boldsymbol{a} \mapsto \{\boldsymbol{a}\}$ and hyperedges $\overline{x} \mapsto \{\boldsymbol{b}\}$ where $\boldsymbol{b}$ is some atom with $\overline{x} \subseteq \text{var}(\boldsymbol{b})$. Note that, since under this definition $|\lambda(n)| = 1$ for all nodes, this indeed yields a GHD of width 1.

Proposition 3.4. *If there exists a width-one GHD for set of atoms A, then there also exists a generalized join tree for A. Consequently, a CQ Q is acyclic iff at(Q) has a generalized join tree.*

Crux. A *traditional join tree* for A is a width-one GHD T for A in which all the nodes are atoms in A (no hyperedges allowed). It is well-known that a width-one GHD exists for A if, and only if, a traditional join tree T exists for A [19]. It hence suffices to show that every traditional join tree T for A can be transformed into a generalized join tree T' for A. We do this by this by recursively applying the following transformation rule to nodes in T, starting at the root:

Let n be the current node being transformed. If hyperedge $\text{var}(n)$ *is not yet in T', then add* $\text{var}(n)$ *to T' and add an edge from* $\text{var}(n)$ *to* $\text{var}(p)$ *with p the parent of n in T. Finally (and even if* $\text{var}(n)$ *were already in T'), add n (which is an atom) to T and add an edge from n to* $\text{var}(n)$*. Then, recursively apply this procedure to each child of n in T.*

To illustrate, if we apply this procedure to traditional join tree T_1 of Figure 2 then we obtain the following generalized join tree.



It is a standard exercise to show that this transformation indeed always yields a generalized join tree. □

B. PROOFS FROM SECTION 4.3

Lemma 4.11. *If $\vec{t} \in \Delta\Gamma_n(db, u)$ then $\vec{t} \in \Delta\Psi_c(db, u)$ for some guard $c \in \text{grd}(n)$. Moreover, if $\vec{t} \in \Delta\Lambda_n(db, u)$ then either (1) $\vec{t} \in \Delta\Psi_c(db, u)$ for some guard $c \in \text{grd}(n)$ or (2) $\vec{t} \in (\Gamma_n(db) \ltimes \Delta\Psi_c(db, u))$ for some child $c \in \text{ng}(n)$.*

Proof. We only show the reasoning when $\vec{t} \in \Delta\Lambda_n(db, u)$. The reasoning when $\vec{t} \in \Delta\Gamma_n(db, u)$ is similar.

Suppose that $\vec{t} \in \Delta\Lambda_n(db, u)$. By definition, $\Lambda_n := \Gamma_n \bowtie \bowtie_{c \in \text{ng}(n)} \Psi_c$. By definition of join tree, $\text{grd}(n)$ is non-empty. If $\text{ng}(n)$ is empty, then in particular, $\vec{t} \in \Delta\Lambda_n(db, u) = \Delta\Gamma_n(db, u)$. In that case, by the first part of the lemma, we hence obtain $\vec{t} \in \Delta\Psi_c(db, u)$ for some $c \in \text{grd}(n)$. It remains to confirm the result when $\text{ng}(n)$ is non-empty. Hereto, first observe that taking deltas distributes over joins as follows.

$$\begin{aligned}\Delta\big(r(\overline{x}) \bowtie s(\overline{y})\big)(db, u) = {} & \Delta r(\overline{x})(db, u) \bowtie s(\overline{y})(db) \\ & + (\Delta r(\overline{x})(db, u) \bowtie \Delta s(\overline{y})(db, u) \\ & + (r(\overline{x})(db, u) \bowtie \Delta s(\overline{y})(db, u)\end{aligned}$$

By application of this equality to $\Delta\Lambda_n(db, u)$, we obtain that there are three cases possible.

- Case $\vec{t} \in \Delta\Gamma_n(db, u) \bowtie \big(\bowtie_{c \in \text{ng}(n)} \Psi_c\big)(db)$. Then, $\vec{t} \in \Delta\Gamma_n(db, u)$ since Γ_n has the same schema as Λ_n. By the first part of the lemma, we hence obtain $\vec{t} \in \Delta\Psi_c(db, u)$ for some $c \in \text{grd}(n)$.
- The case $\vec{t} \in \Delta\Gamma_n(db, u) \bowtie \Delta\big(\bowtie_{c \in \text{ng}(n)} \Psi_c\big)(db, u)$ is similar.
- Case $\vec{t} \in \Gamma_n(db) \bowtie \Delta\big(\bowtie_{c \in \text{ng}(n)} \Psi_c\big)(db, u)$. Then in particular, $\vec{t} \in \Gamma_n(db)$. Moreover, $\vec{t}[\bigcup_{c \in \text{ng}(n)} \text{pvar}(c)]$ is in $\Delta\big(\bowtie_{c \in \text{ng}(n)} \Psi_c\big)(db, u)$. Then by application of the above distribution of delta's over joins on expression $\Delta\big(\bowtie_{c \in \text{ng}(n)} \Psi_c\big)(db, u)$ we obtain that there is at least one $c \in \text{ng}(n)$ such that $\vec{t}[\text{pvar}(c)] \in \Delta\Psi_c(db, u)$. Therefore, $\vec{t} \in \text{supp}(\Gamma_c(db) \ltimes \Delta\Psi_c(db, u))$, as desired. □

C. PROOFS FROM SECTION 4.6

Proposition 4.22. *Under the above-mentioned hypotheses, a DCLR exists for CQ Q if, and only if, Q is free-connex acyclic.*

Crux. The if direction follows from all of our results so far. For the only if direction, assume that query Q is not free-connex acyclic and suppose that a DCLR exists for query Q. In particular, we can compute, for every database db a data structure $\mathbb{D}$ that represents $Q(db)$, for every database db. Let U be the algorithm that maintains these datastructures under updates and let ϵ be the $DCLR$ that represents the empty query result (which is obtained when Q is evaluated on the empty database). Then, starting from ϵ, $U(\epsilon, db)$ must construct a DCLR in time $O(\|\epsilon\| + \|db\|) = O(\|db\|)$ since ϵ is constant. Now enumerate $Q(db)$ from $U(\epsilon, db)$ with constant delay but do not output tuple multiplicities. This enumerates $Q(db)$ evaluated under set semantics. Then $Q \in \text{CD} \circ \text{LIN}$, contradicting Brault-Baron [10]. □

Proposition 4.25. *If a CQ Q is q-hierarchical, then it has a join tree which is both simple and compatible.*

Proof. A CQ Q is *connected* if for any two $x, y \in \text{var}(Q)$ there is a path $x = z_0, \ldots, z_l = y$ such that for each $j < l$ there is an atom $\boldsymbol{a}$ in Q such that $\{z_j, z_{j+1}\} \subseteq \text{var}(\boldsymbol{a})$. It is a standard observation that every CQ can be written as a join $Q_1 \bowtie \cdots \bowtie Q_k$ of connected CQs with pairwise disjoint sets of output variables. Call these Q_i the connected components of Q. Berkholz et al. show that CQ Q is hierarchical if and only if every connected component Q_i of Q has a q-tree, which is defined as follows.

Definition C.1. Let Q_i be a connected CQ. A q-tree for Q_i is a rooted directed tree $F_{Q_i} = (V, E)$ with $V = \text{var}(Q)$ s.t. (1) for all atoms $\boldsymbol{a}$ in Q_i the set $\text{var}(\boldsymbol{a})$ forms a directed path in F_Q starting at the root, and (2) if $out(Q_i) \neq \emptyset$, then $out(Q_i)$ is a connected subset in F_{Q_i} containing the root.

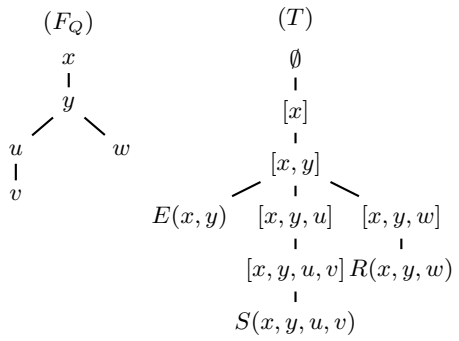


Figure 7: Illustration of the proof of Proposition 4.25

To show the proposition, assume that CQ Q is hierarchical. From the q-trees for the connected components of Q we can construct a simple join tree T for Q that is compatible with Q, as follows. For ease of exposition, let us assume that Q has a single connected component; the general case is similar. Let F_Q be the q-tree for Q. Then, for every node x in F_Q, define $p(x)$ to be set of variables that occur in the unique path from x to the root in F_Q. In particular, $p(x)$ contains x. By definition of q-trees, $p(x)$ must be a partial hyperedge of A, the set of atoms in Q. Construct T as follows. Initially, T contains only the empty hyperedge $\emptyset$. For all variables $x \in \text{var}(Q)$, add hyperedge $p(x)$ to T. For every edge $x \to y$ in F_Q, add an edge $p(x) \to p(y)$ to T. If x is the root in F_Q, then also add the edge $p(x) \to \emptyset$ to the root $\emptyset$ in T. Next, add all atoms of Q to T, and for each atom $\boldsymbol{a}$, add an edge from $\boldsymbol{a}$ to the hyperedge h in T with $h = \text{var}(\boldsymbol{a})$. (This hyperedge has been generated by $p(x)$ with x the variable in $\text{var}(\boldsymbol{a})$ that is the lowest among all variables of $\text{var}(\boldsymbol{a})$ in F_Q). Figure 7 illustrates this construction for $Q = \pi_{x,y,u}(E(x,y) \bowtie R(x,y,w) \bowtie S(x,y,u,v))$ and the q-tree F_Q shown in Figure 7. Note that in this example, T is indeed a simple generalized join tree. It can be shown that this is always the case. It remains to show that T is compatible with Q. To do so, observe that, by definition of q-tree, $\mathit{out}(Q)$ is a connected subset of F_Q that contains the root. Then $N = \{\emptyset\} \cup \{p(x) \mid x \in \mathit{out}(Q)\}$ must be a connex subset of T with $\text{var}(N) = \mathit{out}(Q)$, as desired. □

Proposition 4.26. *If a CQ Q has a join tree T which is both simple and compatible with Q, then Q is q-hierarchical.*

Proof. Assume that T is a simple join tree for Q that is also compatible with Q. We first show that Q is hiearchical. Let x and y be two variables in Q. If $\mathit{at}(x) \cap \mathit{at}(y) = \emptyset$ we are done. Hence, assume $\mathit{at}(x) \cap \mathit{at}(y) \neq \emptyset$. Let $\boldsymbol{c} \in \mathit{at}(x) \cap \mathit{at}(y)$. We need to show that either $\mathit{at}(x) \subseteq \mathit{at}(y)$ or $\mathit{at}(y) \subseteq \mathit{at}(x)$. Assume for the purpose of contradiction that neither holds. Then there exists $\boldsymbol{a} \in \mathit{at}(x) \setminus \mathit{at}(y)$ and similarly an atom $\boldsymbol{b} \in \mathit{at}(y) \setminus \mathit{at}(x)$. Since T is a join tree, and since x occurs both in $\boldsymbol{a}$ and $\boldsymbol{c}$, we know that x must occur in every node on the unique undirected path between $\boldsymbol{a}$ and $\boldsymbol{c}$ in T. In particular, let n be the least common ancestor of $\boldsymbol{a}$ and $\boldsymbol{c}$. Then $x \in \text{var}(n)$. Similarly, y must occur in every node on the unique undirected path between $\boldsymbol{b}$ and $\boldsymbol{c}$ in T. In particular, let m be the least common ancestor of $\boldsymbol{b}$ and $\boldsymbol{c}$. Then $y \in \text{var}(m)$. Now there are two possibilities. Either (1) n is an ancestor of m. But then, since T is simple, $x \in \text{var}(n) \subseteq \text{var}(m)$. Since $\boldsymbol{b}$ is a descendant of m then by simplicity of T hence $x \in \text{var}(n) \subseteq \text{var}(m) \subseteq \text{var}(\boldsymbol{b})$. This contradicts our assumption that $\boldsymbol{b} \in \mathit{at}(y) \setminus \mathit{at}(x)$. Otherwise, (2) m is an ancestor of m and we similarly obtain a contradiction to our assumption that $\boldsymbol{a} \in \mathit{at}(x) \setminus \mathit{at}(y)$.

It remains to show q-hierachicality. Hereto, assume that $\mathit{at}(x) \subsetneq \mathit{at}(y)$ and $x \in \mathit{out}(Q)$. We need to show that $y \in \mathit{out}(Q)$. Let $\boldsymbol{a} \in \mathit{at}(x)$ and let $\boldsymbol{b} \in \mathit{at}(y) \setminus \mathit{at}(x)$. In particular, $\boldsymbol{a}$ contains both x and y, while $\boldsymbol{b}$ contains only y. From compatibility of T with Q, it follows that there is a connex subset N of T such that $\text{var}(N) = \mathit{out}(Q)$. Let n be the lowest ancestor node of $\boldsymbol{a}$ in N that contains x. Because T is simple, all descendants of n hence also have x. As a consequence, $\boldsymbol{b}$ cannot be a descendant of n. Since $\boldsymbol{a}$ and $\boldsymbol{b}$ share y, this implies that the unique undirected path between $\boldsymbol{a}$ and $\boldsymbol{b}$ must pass through n. Because all nodes on this path must share all variables in common between $\boldsymbol{a}$ and $\boldsymbol{b}$, it follows that $y \in \text{var}(n) \subseteq \text{var}(N) = \mathit{out}(Q)$. □

Proposition 4.27. *Let Q be the hierarchical join query $R(x,y,z) \bowtie S(x,y,u) \bowtie T(y,v,w) \bowtie U(y,v,p)$. Every simple width-one GHD for Q has at least one partial hyperedge.*

Proof. Let T be a simple width-one GHD for Q and assume, for the purpose of contradiction that T contains only atoms and full hyperedges. T's nodes are hence elements of $\{R(x,y,z), S(x,y,u), T(y,v,w), U(y,v,p), [x,y,z], [x,y,u], [y,v,w], [y,v,p]\}$. Partition this set into

$$XY = \{R(x,y,z), S(x,y,u), [x,y,z], [x,y,u]\}$$
$$YV = \{T(y,v,w), U(y,v,p), [y,v,w], [y,v,p]\}.$$

Now consider the unique undirected path $m, n_1, n_2, \ldots, n_k, p$ between $m = R(x,y,z)$ and $p = T(y,v,w)$. There are two possibilities: either this undirected path shows that some node in XY is a parent of a node in YV, or it shows that some node in YV is a parent of some node in XY. In either case, hierarchicality is violated since nodes in XY all have variable x while nodes in YV don't and, conversely, nodes in YV all have variable v while nodes in XY don't. □

D. QUERIES

Full join queries

```
FQ1
SELECT *  FROM  orders o, lineitem l, part p , partsupp ps
WHERE  o.orderkey = l.orderkey,  AND  l.partkey = p.partkey
AND  l.partkey = ps.partkey  AND  l.suppkey = ps.suppkey

FQ2
SELECT *  FROM  lineitem l, orders, customer c, part p , nation n
WHERE  l.orderkey = o.orderkey  AND  o.custkey = c.custkey
AND  l.partkey = p.partkey  AND  c.nationkey = n.nationkey

FQ3
SELECT *  FROM  orders o, lineitem l,
partsupp ps, supplier s, customer c
WHERE  o.orderkey = l.orderkey  AND
l.suppkey = ps.suppkey  AND
l.suppkey = s.suppkey  AND  o.custkey = c.custkey

FQ4
SELECT *  FROM  lineitem l, supplier s, partsupp ps
WHERE  l.suppkey = s.suppkey
AND  l.suppkey = ps.suppkey

FQ5
SELECT * SELECT *  FROM  date_dim dd, store_sales ss, item i
WHERE  ss.s_item_sk = i.i_item_sk
AND  ss.s_date_sk = dd.d_date_sk
```

TPC-H

```
Q1
SELECT l_returnflag, l_linestatus, SUM(l_quantity)
AS sum_qty, SUM(l_extendedprice) AS sum_base_price,
SUM(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
SUM(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS
sum_charge, AVG(l_quantity) AS AVG_qty, AVG(l_extendedprice)
AS AVG_price, AVG(l_discount) AS AVG_disc, count(*)
AS count_order
FROM  lineitem  WHERE
l_shipdate <= date '1998-12-01' - interval '108' day
group by  l_returnflag,l_linestatus

Q3
SELECT l_orderkey,SUM(l_extendedprice * (1 - l_discount))
AS revenue,o_orderdate,o_shippriority
FROM customer,orders,lineitem  WHERE c_mktsegment = 'AUTOMOBILE'
AND  c_custkey = o_custkey AND  l_orderkey = o_orderkey
AND  o_orderdate < date '1995-03-13' AND
l_shipdate > date '1995-03-13'
group by l_orderkey,o_orderdate,o_shippriority

Q4
SELECT o_orderpriority, count(*) AS order_count
FROM orders  WHERE o_orderdate >= date '1995-01-01' AND
o_orderdate < date '1995-01-01' + interval '3' month
AND  exists ( SELECT *  FROM  lineitem  WHERE
l_orderkey = o_orderkey AND  l_commitdate < l_receiptdate)
group by o_orderpriority

Q6
SELECT SUM(l_extendedprice * l_discount) AS revenue
FROM lineitem
WHERE  l_shipdate >= date '1994-01-01' AND
l_shipdate < date '1994-01-01' + interval '1' year
AND  l_discount between 0.06 - 0.01  AND
0.06 + 0.01 AND  l_quantity < 24;

Q9
SELECT nation,o_year,SUM(amount) AS sum_profit
FROM ( SELECT n_name
AS nation,extract(year  FROM  o_orderdate) AS o_year,
l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity
AS amount
FROM  part,supplier,lineitem,partsupp,
orders,nation WHERE  s_suppkey = l_suppkey
AND ps_suppkey = l_suppkey  AND
ps_partkey = l_partkey  AND  p_partkey = l_partkey
AND  o_orderkey = l_orderkey
AND  s_nationkey = n_nationkey
AND  p_name like '%dim%') AS profit
group by nation,o_year

Q12
SELECT l_shipmode, SUM(case when o_orderpriority = '1-URGENT'
or o_orderpriority = '2-HIGH'
then 1 else 0 end) AS high_line_count,
SUM(case when o_orderpriority <> '1-URGENT'
AND  o_orderpriority <> '2-HIGH' then 1 else 0 end)
AS low_line_count
FROM orders,lineitem
WHERE o_orderkey = l_orderkey AND
l_shipmode in ('RAIL', 'FOB') AND
l_commitdate < l_receiptdate AND
l_shipdate < l_commitdate AND
l_receiptdate >= date '1997-01-01' AND
l_receiptdate < date '1997-01-01' + interval '1' year
group by l_shipmode

Q13'
SELECT c_count, COUNT(*) AS custdist
FROM  (
SELECT c.custkey AS c_custkey, COUNT(o.orderkey) AS c_count
FROM  customer c, orders o
WHERE  c.custkey = o.custkey
AND  (o.comment NOT LIKE '%special%requests%')
group by  c.custkey) c_orders
group by  c_count;

Q16'
SELECT p_brand, p_type, p_size, count(distinct ps_suppkey)
as supplier_cnt
FROM partsupp, part
WHERE  p_partkey = ps_partkey AND p_brand <> 'Brand#34'
AND p_type not like 'LARGE BRUSHED%'
AND p_size in (48, 19, 12, 4, 41, 7, 21, 39)
AND ps_suppkey in ( SELECT s_suppkey FROM supplier
WHERE s_comment not like '%Customer%Complaints%' )
GROUP BY  p_brand, p_type, p_size

Q18
SELECT c_name,c_custkey,o_orderkey,o_orderdate,
o_totalprice,SUM(l_quantity)
FROM customer,orders,lineitem
WHERE o_orderkey in (SELECT l_orderkey
FROM  lineitem
group by  l_orderkey having SUM(l_quantity) > 314)
AND  c_custkey = o_custkey AND  o_orderkey = l_orderkey
group by
c_name,c_custkey,o_orderkey,o_orderdate,o_totalprice
```

TPC-DS

```
Q3
SELECT dt.d_year, i.i_brand _id, i.i_brand ,
SUM(ss.ss_ext_sales_price) AS sum_agg
FROM  date_dim dt, store_sales ss, item i
WHERE  dt.d_date_sk = ss.ss_sold_date_sk
AND  ss.ss_item_sk = i.i_item_sk
AND  dt.d_moy = 12
AND  i.i_manufact_id = 436
group by dt.d_year, i.i_brand , i.i_brand _id;

Q7
SELECT i.i_item_id, AVG(ss.ss_quantity)
AS agg1, AVG(ss.ss_list_price)
AS agg2, AVG(ss.ss_coupon_amt) AS agg3,
AVG(ss.ss_sales_price) AS agg4
FROM  store_sales ss, customer_demographics cd,
date_dim d, item i, promotion p
WHERE  ss.ss_item_sk = i.i_item_sk
AND  ss.ss_sold_date_sk = d.d_date_sk
AND  ss.ss_cdemo_sk = cd.cd_demo_sk
AND  ss.ss_promo_sk = p.p_promo_sk
AND  cd.cd_gender = 'F'
AND  cd.cd_marital_status = 'W'
AND  (p.p_channel_email = 'N' OR p.p_channel_event = 'N')
AND  d.d_year = 1998
group by i.i_item_id;

Q19
SELECT i.i_brand_id, i.i_brand , i.i_manufact_id,
i.i_manufact, SUM(ss.ss_ext_sales_price) AS ext_price
FROM  date_dim dd, store_sales ss, item i,
customer c, customer_address ca, store s
WHERE  dd.d_date_sk = ss.ss_sold_date_sk
AND  ss.ss_item_sk = i.i_item_sk
AND  i.i_manager_id = 7
AND  dd.d_moy = 11
AND  dd.d_year = 1999
AND  ss.ss_customer_sk = c.c_customer_sk
AND  c.c_current_addr_sk = ca.ca_address_sk
AND  ss.ss_store_sk = s.s_store_sk
group by
i.i_br  , i.i_br, _id, i.i_manufact_id, i.i_manufact;

Q22
SELECT i.i_product_name, i.i_brand ,
i.i_class, i.i_category,
SUM(inv.inv_quantity_on_hand) AS qoh
FROM  date_dim dd, inventory inv, item i,
warehouse wh
WHERE  dd.d_date_sk = inv.inv_date_sk
AND  inv.inv_item_sk = i.i_item_sk
AND  inv.inv_warehouse_sk = wh.w_warehouse_sk
AND  dd.d_month_seq between 1193  AND  1204
group by
i.i_product_name, i.i_brand , i.i_clASs, i.i_category;
```