Efficient Query Processing for Dynamically Changing Datasets [*]

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

Martín Ugarte
PUC Chile
martinugarte@uc.cl

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

Hannes Voigt
Neo4J
hannes.voigt@neo4j.com

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

ABSTRACT

The ability to efficiently analyze changing data is a key requirement of many real-time analytics applications. Traditional approaches to this problem were developed around the notion of Incremental View Maintenance (IVM), and are based either on the materialization of subresults (to avoid their recomputation) or on the recomputation of subresults (to avoid the space overhead of materialization). Both techniques are suboptimal: instead of materializing results and subresults, one may also maintain a data structure that supports efficient maintenance under updates and from which the full query result can quickly be enumerated. In two previous articles, we have presented algorithms for dynamically evaluating queries that are easy to implement, efficient, and can be naturally extended to evaluate queries from a wide range of application domains. In this paper, we discuss our algorithm and its complexity, explaining the main components behind its efficiency. Finally, we show experiments that compare our algorithm to a state-of-the-art (Higher-order) IVM engine, as well as to a prominent complex event recognition engine. Our approach outperforms the competitor systems by up to two orders of magnitude in processing time, and one order in memory consumption.

1 Introduction

The ability to efficiently analyze changing data is a key requirement of many real-time analytics applications like Stream Processing [20], Complex Event Recognition [9], Business Intelligence [17], and Machine Learning [22].

In this context, we tackle the problem of *dynamic query evaluation*, where a given query Q has to be evaluated against a database that is constantly changing. Concretely, when database db is updated to database $db + u$ under update u, the objective is to efficiently compute $Q(db + u)$, taking into consideration that $Q(db)$ was already evaluated and re-computations could be avoided.

Dynamic query evaluation is of utmost importance if response time requirements for queries under concurrent data updates have to be met or if data volumes are so large that full re-evaluation of queries based on raw data is prohibitive.

The following example illustrates our setting. Assume that we wish to detect potential credit card fraud. Credit card transactions specify their timestamp (ts), account number (acc), and amount ($amnt$). A typical fraud pattern is that, in a short period of time, a criminal tests a stolen credit card with a few small purchases to then make larger purchases (cf. [18]). Assuming that the short period of time is 1 hour, this pattern could be detected by dynamically evaluating the query in Figure 1. Queries like this may exhibit arbitrary local predicates and multi-way joins with equality as well as inequality predicates. Traditional techniques to process such queries dynamically can be categorized in two approaches: *relational* and *automaton-based*. We outline the core principles of relational approaches in the following and refer to [13] for an in-depth discussion of the drawbacks of automaton-based approaches.

Relational approaches such as [2, 10, 16] build upon the technique of *Incremental View Maintenance (IVM)* [7]. To process a query Q over a database db, IVM techniques materialize the output $Q(db)$ and evaluate *delta queries* [10]. Upon update u, delta queries use db, u, and the materialized $Q(db)$ to compute the set of tuples to add/delete from $Q(db)$ in order to obtain $Q(db + u)$. If u is small with respect to the database db, this is expected to be faster than recomputing $Q(db + u)$ from scratch. To further speed up dynamic query processing, also the result of some subqueries of Q may be redundantly materialized. This approach is known as Higher-Order IVM (HIVM) [15, 16].

Unfortunately, (H)IVM shows a serious drawback in terms of additional memory overhead, which quickly becomes prohibitive for interactive data analytics scenarios: materialization of $Q(db)$ requires $\Omega(|Q(db)|)$ space, where $|db|$ denotes the size of db. Therefore, when $Q(db)$ is large, which is often the case in data preparation scenarios for training statistical models, 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

[*] This paper surveys the results of two papers: "The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates" [12], which was published in SIGMOD Conference 2017 and "Conjunctive Queries with Inequalities Under Updates" [13], published in PVLDB 11(7), 2018. During the work described, Martín Ugarte was affiliated to ULB and Hannes Voight to TU Dresden.

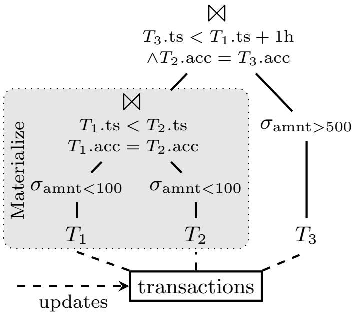

Figure 1: Example query for detecting fraudulent credit card activity.

materializes the result of Q but also the results of some subqueries. In fact some of these subresults can be partial join results, which can be larger than both db and $Q(db)$. For example, in our fraud query, HIVM would materialize the results of the join in the shaded area in Figure 1. Intuitively, this join builds the table of all pairs of *small* transactions that could be part of a credit card fraud if a third relevant transaction occurs. Therefore, if we assume that there are N small transactions in the time window, all of the same account, this materialization will take $\Theta(N^2)$ space. This naturally becomes impractical when N grows.

While these problems are inherent to (H)IVM methods, they can be avoided by taking a different approach to dynamic query evaluation: instead of materializing $Q(db)$ we can build a succinct data structure that (1) supports efficient maintenance under updates and (2) *represents* $Q(db)$ in the sense that from it we can *generate* $Q(db)$ as efficiently as if it were materialized. In particular, the representation is equipped with index structures so that we can enumerate $Q(db)$ *with constant delay* [19]: one tuple at a time, while spending only a constant amount of work to produce each new tuple. This makes the enumeration competitive with enumeration from materialized query results.

In essence, we hence separate dynamic query processing into two stages: (1) an update stage where we only maintain under updates the (small) information that is necessary for result enumeration and (2) an enumeration stage where the query result is efficiently enumerated.

In our work, which is documented in detail in [12] and [13], we are concerned with designing a practical family of algorithms for dynamic query evaluation based on this idea for queries featuring both equi-joins and inequality joins, as well as certain forms of aggregation. Our main insight is that, for acyclic conjunctive queries, such algorithms can naturally be obtained by modifying Yannakakis' seminal algorithm for processing acyclic joins in the static setting [23].

In a first step, we address the problem of efficiently evaluating acyclic aggregate-join queries by providing the *Dynamic Yannakakis Algorithm* (DYN) [12]. The representation of query results that underlies this algorithm has several desirable properties:

- (P_1) It allows to enumerate $Q(db)$ with constant delay.
- (P_2) For any tuple $\vec{t}$, it can be used to check whether $\vec{t} \in Q(db)$ in constant time.
- (P_3) It requires only $\mathcal{O}(|db|)$ space and is hence independent of the size of $Q(db)$.
- (P_4) it features efficient maintenance under updates: given update u to db, we can update the representation of $Q(db)$ to a representation of $Q(db + u)$ in time $\mathcal{O}(|db| + |u|)$. In contrast, (H)IVM may require $\Omega(|u| + |Q(db+u)|)$ time in the worst case. For the subclass of q-hierarchical queries [4], our update time is $\mathcal{O}(|u|)$.

Based on this technique to dynamically process queries with equi-joins, we provide the core intuiton of a generalization of the Dynamic Yannakakis Algorithm to conjunctive queries with arbitrary θ-joins. We show that, in the specific case of inequality joins, this generalization improves the state of the art for dynamically processing inequality joins by performing consistently better, with up to two orders of magnitude improvements in processing time and one order in memory consumption.

It is important to note that we consider query evaluation in main memory and measure time and space under data complexity [21]. 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.

2 Preliminaries

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

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

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

Example 2.1. The following query is a GCQ.

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

Intuitively, the query asks to take the natural join of $r(x, y)$ and $s(y, z, w)$, form the cartesian product with $t(u, v)$, and subsequently select those tuples that satisfy $x < z$ and $w < u$.

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

Semantics. We evaluate GCQs over Generalized Multiset Relations (GMRs for short) [12, 15, 16]. Let $\text{dom}(x)$ denote the domain of variable x. As usual, a tuple over $\overline{x}$ is a mapping $\vec{t}$ that assigns a value from $\text{dom}(x)$ to every $x \in \overline{x}$. A GMR R over $\overline{x}$ is a function $R\colon \mathbb{T}[\overline{x}] \to \mathbb{Z}$ mapping tuples over $\overline{x}$ to integers such that $R(\vec{t}) \neq 0$ for finitely many tuples $\vec{t}$. Here, $\mathbb{T}[\overline{x}]$ denotes the set of all tuples over x. In contrast to classical multisets, the multiplicity of a tuple in a GMR can

R

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

S

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

T

u	v	$\mathbb{Z}$
4	5	−4
2	1	6
1	4	3

$S \bowtie T$

u	v	$\mathbb{Z}$
4	5	−20
1	4	6

$\pi_y(R)$

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

$S + T$

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

$S - T$

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

$\sigma_{y<u}(R \bowtie S)$

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

Figure 2: Operations on GMRs.

hence be negative, allowing to treat insertions and deletions uniformly. We write $var(R)$ for $\overline{x}$; $\text{supp}(R)$ for the finite set of all tuples with non-zero multiplicity in R; $\vec{t} \in R$ to indicate $\vec{t} \in \text{supp}(R)$; and $|R|$ for $|\text{supp}(R)|$. A GMR R is *positive* if $R(\vec{t}) > 0$ for all $\vec{t} \in \text{supp}(R)$. The operations of GMR union ($R + S$), minus ($R - S$), projection ($\pi_{\overline{z}} R$), natural join ($R \bowtie T$) and selection ($\sigma_P(R)$) are defined similarly as in relational algebra with multiset semantics. Figure 2 illustrates these operations; see [12, 16] for formal semantics.

A *database* over a set $\mathcal{A}$ of atoms is a function db that maps every atom $r(\overline{x}) \in \mathcal{A}$ to a positive GMR $db_{r(\overline{x})}$ over $\overline{x}$. We write $|db|$ for $\sum_{r(\overline{x}) \in \mathcal{A}} |db_{r(\overline{x})}|$. Given a database db over the atoms occurring in query Q, the evaluation of Q over db, denoted $Q(db)$, is the GMR over $\overline{y}$ constructed in the expected way: take the natural join of all GMRs in the database, do a selection over the result w.r.t. each predicate, and finally project on $\overline{y}$. It is instructive to note that after evaluation, each result tuple has an associated multiplicity that counts the number of derivations for the tuple. In other words, the query language has built-in support for `COUNT` aggregations. We note that, in their full generality, GMRs can carry multiplicities that are taken from an arbitrary algebraic semiring structure (cf., [15]), which can be useful to describe the computation of more advanced aggregations over the result of a GCQ [1]. To keep the notation and discussion simple we fix the ring $\mathbb{Z}$ of integers throughout the paper, but our results generalize to arbitrary semirings and their associated aggregations.

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.

Computational Model. We focus on dynamic query evaluation in main-memory. We assume a model of computation where the space used by tuple values and integers, the time of arithmetic operations on integers, and the time of memory lookups are all $\mathcal{O}(1)$. We further assume that every GMR R can be represented by a data structure that allows (1) enumeration of R with constant delay (as defined in Section 3); (2) multiplicity lookups $R(\vec{t})$ in $\mathcal{O}(1)$ time given $\vec{t}$; (3) single-tuple insertions and deletions in $\mathcal{O}(1)$ time; while (4) having size that is proportional to $|R|$. 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 $\mathcal{O}(1)$ time a pointer to the GMR $R \ltimes \vec{s}$, which is the GMR over $\overline{x}$ consisting of all tuples that project to $\vec{s}$:

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

Moreover, we assume that single-tuple insertions and deletions to R can be reflected in the index in $\mathcal{O}(1)$ time and that an index takes space linear in $|R|$. Essentially, our assumptions amount to perfect hashing of linear size [8]. Although this does not directly match a realistic setting, it is well known that complexity results for this model can be translated, through amortized analysis, to average complexity in real-life implementations [8].

3 Dynamic Yannakakis

In this section we formulate Dyn, a dynamic version of the Yannakakis algorithm [23], that focuses on the evaluation of *CQs*. How to deal with θ-joins is discussed in Section 4.

3.1 Intuition

A data structure D supports *enumeration* of a set E if there is a routine ENUM such that ENUM(D) outputs each element of E exactly once. Such enumeration occurs with delay d if the time until the first output; the time between any two consecutive outputs; and the time between the last output and the termination of ENUM(D), are all bounded by d. D supports enumeration of a GMR R if it supports enumeration of the set $E_R = \{(\vec{t}, R(\vec{t})) \mid \vec{t} \in \text{supp}(R)\}$. When evaluating a GCQ Q over a database db, we will be interested in representing the elements of $Q(db)$ by means of a data structure D_{db}, such that we can enumerate $Q(db)$ from D_{db}. If, for every db, the delay to enumerate $Q(db)$ from D_{db} is independent of $|db|$ then we say that the enumeration occurs *with constant delay* [19].

As a trivial example of constant delay enumeration (CDE for short) 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. Since array indexation is a $\mathcal{O}(1)$ operation, this gives constant delay. This example shows that CDE of $Q(db)$ can always be done naively by materializing $Q(db)$ in an in-memory array. Unfortunately, this requires memory proportional to $|Q(db)|$ which, depending on Q, can be of size polynomial in $|db|$. We hence desire other data structures to represent $Q(db)$ using less space, while still allowing CDE.

To understand how this can be done, it is instructive to consider a simple binary join $Q = R(x, y) \bowtie S(y, z)$ and analyze why traditional join processing algorithms do not yield CDE. 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 y (i.e. the hash table) has already been computed. Now observe that, when iterating over S to probe the index, we may have to visit an unbounded number of S-tuples that do not join with any R-tuple. Consequently, the delay between consecutive output tuples may be as large as $|S|$, which is not constant. A similar analysis shows that other join algorithms, such as the sort-merge join, do not yield enumeration with constant delay.

How can we obtain CDE for $R(x, y) \bowtie S(y, z)$? Intuitively speaking, if we can ensure to only iterate over those S-tuples that have matching R-tuples, we trivially obtain constant delay since then every probe will yield a new output tuple. As such, the key is to first compute $Y = \pi_y(R) \bowtie \pi_y(S)$ and index both R and S on y. We can then iterate over the elements of Y, probing both R and S in each iteration to generate the output with constant delay. In the presence of updates, this means that we only need to maintain Y, as well as the indexes on R and S—all of which are of size linear in *db* and can be maintained efficiently.

3.2 The Algorithm

To extend the intuition of Section 3.1 from a binary join to general CQs that feature both multiway equi-joins and projections, we need to maintain for all the relations that are used as *probe* relations in a join, the set of tuples that will match the corresponding build relation(s). Of course, we also need to decide in what order we will join the relations, since this determines the auxiliary sets of tuples (like Y above) that we need to maintain. For DYN, this query plan is specified by means of a pair (T, N) called a *GJT pair.*

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

Definition 3.1. A *GJT pair* is a tuple (T, N) with T a *generalized join tree* and N a *sibling-closed connex subset* of T. A generalized join tree (GJT) is a node-labeled directed tree $T = (V, E)$ such that:
- T is binary: every node has at most two children.
- Every leaf is labeled by an atom.
- Every interior node n is labeled by a hyperedge and has at least one child c such that $\mathit{var}(n) \subseteq \mathit{var}(c)$. Such a child is called a *guard* of n.
- Whenever the same variable x occurs in the label of two nodes m and n of T, then x occurs in the label of each node on the unique path linking m and n.

A *simple GJT* is a GJT where $\mathit{var}(n) \subseteq \mathit{var}(c)$ for every node n with child c, i.e., a GJT where every child is a guard of its parent. A *connex subset* of T is a set $N \subseteq V$ that includes the root of T such that the subgraph of T induced by N is a tree. N is *sibling-closed* if for every node $n \in N$ with a sibling m in T, m is also in N. The *frontier* of a connex set N is the subset $F \subseteq N$ consisting of those nodes in N that are leaves in the subtree of T induced by N.

Figure 3 shows a GJT pair (T_1, N_1) and a GJT T_2. T_1 is simple, but T_2 is not since $t(x, u)$ is not a guard of $\{x, y\}$. The set $N_1 = \{\{x\}, \{x, y\}, t(x, u)\}$, highlighted in gray, is a sibling-closed connex subset of T_1, and its frontier is $\{\{x, y\}, t(x, u)\}$.

Definition 3.2. Let (T, N) be a GJT pair and assume that $\{\!|r_1(\overline{x_1}), \dots, r_n(\overline{x_n})|\!\}$ is the multiset of atoms occurring as labels in the leaves of T. Then the query associated to T is the full join $\mathcal{Q}[T] = (r_1(\overline{x_1}) \bowtie \cdots \bowtie r_n(\overline{x_n}))$ and the query associated to (T, N) is the CQ $\mathcal{Q}[T, N] = \pi_{\mathit{var}(N)}(\mathcal{Q}[T])$.

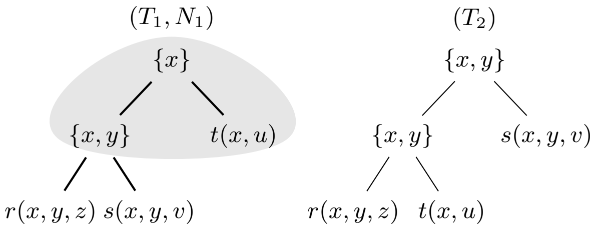

Figure 3: Two example GJTs.

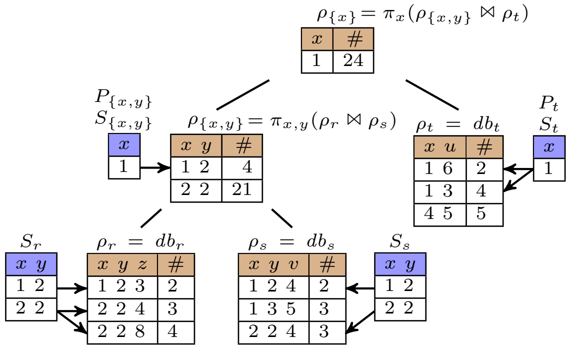

Figure 4: (T_1, N_1)-representation for the database *db* specified by the GMRs depicted at the leaves.

The data structure. Following the intuition of Section 3.1, a GJT pair (T, N) acts as query plan by which DYN processes $\mathcal{Q}[T, N]$ dynamically. In particular, the GJT T specifies the data structure to be maintained and drives the processing of updates, while the connex set N drives the enumeration of query results. The data structure itself is defined next.

Definition 3.3. Let (T, N) be a GJT pair and let *db* be a database over $\mathit{at}(Q)$. The *T-reduct* (or *semi-join reduction*) of *db* is a collection ρ of GMRs, one GMR ρ_n for each node $n \in T$, defined inductively as follows:
- if $n = r(x)$ is an atom, then $\rho_n = db_{r(x)}$
- if n has a single child c, then $\rho_n = \pi_{\mathit{var}(n)}\rho_c$
- otherwise, n has two children c_1 and c_2. In this case we have $\rho_n = \pi_{\mathit{var}(n)}(\rho_{c_1} \bowtie \rho_{c_2})$. Note that, because n has a guard child, this is actually a semijoin.

A T-reduct needs to be augmented by suitable index structures to be used for both enumeration and maintenance under updates. Concretely for each node n with parent p in T, the following indexes are created:
- If n belongs to N, then we store an index P_n of ρ_n on $\mathit{var}(p) \cap \mathit{var}(n)$, called the *parent index* of n.
- If n is a node with a sibling m, then we store an index S_n of ρ_n on $\mathit{var}(n) \cap \mathit{var}(m)$, called the *sibling index* of n.

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

Figure 4 depicts an example (T_1, N_1)-representation ρ for the database *db* composed of the GMRs shown at the leaves of the tree. It is important to observe that the size of this representation for a database *db* can be at most linear in the size of *db*. The reason is that each interior node only does projections or semijoins. Therefore, as illustrated in Figure 4, for each node n there is some descendant atom

Algorithm 1 DYN: Dynamic Yannakakis

```
1:  function ENUM$_{T,N}(\rho)$
2:    for each $\vec{t} \in \rho_{\mathrm{root}(T)}$ do ENUM$_{T,N}(root(T), \vec{t}, \rho)$

3:  function ENUM$_{T,N}(n, \vec{t}, \rho)$
4:    if $n$ is in the frontier of $N$ then yield $(\vec{t}, \rho_n(\vec{t}))$
5:    else if $n$ has one child $c$ then
6:      for each $\vec{s} \in \rho_c \ltimes \vec{t}$ do ENUM$_{T,N}(c, \vec{s}, \rho)$
7:    else $n$ has two children $c_1$ and $c_2$
8:      for each $\vec{t_1} \in \rho_{c_1} \ltimes \vec{t}$ do
9:        for each $\vec{t_2} \in \rho_{c_2} \ltimes \vec{t}$ do
10:         for each $(\vec{s_1}, \mu) \in$ ENUM$_{T,N}(c_1, \vec{t_1}, \rho)$ do
11:           for each $(\vec{s_2}, \nu) \in$ ENUM$_{T,N}(c_2, \vec{t_2}, \rho)$ do
12:             yield $(\vec{s_1} \cup \vec{s_2},\ \mu \times \nu)$

13: procedure UPDATE$_{T,N}(\rho, u)$
14:   for each $n \in$ leafs$(T)$ labeled by $r(\overline{x})$ do
15:     $\Delta_n \leftarrow u_{r(\overline{x})}$
16:   for each $n \in$ nodes$(T) \setminus$ leafs$(T)$ do
17:     $\Delta_n \leftarrow$ empty GMR over $var(n)$
18:   for each $n \in$ nodes$(T)$, traversed bottom-up do
19:     $\rho_n += \Delta_n$
20:     if $n$ has a parent $p$ and a sibling $m$ then
21:       $\Delta_p += \pi_{var(p)}\,(\rho_m \bowtie \Delta_n)$
22:     else if $n$ has parent $p$ then
23:       $\Delta_p += \pi_{var(p)}\,\Delta_n$
```

α (possibly n itself) such that $\mathrm{supp}(\rho_n) \subseteq \mathrm{supp}(\pi_{var(n)}\ db_\alpha)$. Consequently, the indexes are also of size linear in *db*.

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

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

Update processing. To maintain a (T, N)-rep under update u it suffices to traverse the nodes of T in a bottom-up fashion. At each node n we have to compute the update Δ_n to apply to ρ_n and its associated indexes. For leaf nodes, this update is given by the update u itself. For interior nodes, Δ_n can be computed from the update and the original reduct of its children. Algorithm 1 gives the pseudocode. Here, line 21 is then implemented by means of a straightforward hash-join (using the sibling index S_m on ρ_m). As a side effect of modifying ρ the associated indexes are also updated (not shown).

Theorem 3.4. *Let (T, N) be a fixed GJT pair. Given a (T, N)-rep of db with T-reduct ρ,* ENUM$_{T,N}(\rho)$ *enumerates $\mathcal{Q}[T, N](db)$ with constant delay. Moreover,* UPDATE(ρ, u) *updates the (T, N)-rep from a (T, N)-rep of db to a (T, N)-rep of $db + u$ in time $\mathcal{O}(|db| + |u|)$. If T is simple, then the update time is $\mathcal{O}(|u|)$, hence independent of $|db|$.*

Note that UPDATE$_{T,N}$ can be used to build a (T, N)-rep of *db* in time $\mathcal{O}(|db|)$: start from an empty (T, N)-rep (which represents the empty database) and then call UPDATE$_{T,N}(\rho, u)$ with $u = db$. This hence shows that DYN can be used to enumerate $\mathcal{Q}[T, N](db)$ with constant delay after linear time preprocessing.

We also note that if $\vec{t}$ is a tuple over $var(M)$ for some connex subset $M \subseteq N$ of T, then checking whether $\vec{t} \in \pi_{var(M)}\mathcal{Q}[T, N](db)$ can be done in constant time: it suffices to check that $\vec{t}[var(m)] \in \rho_m$ for every $m \in M$ and return true if and only if this is the case. Since T and N are fixed, the size of M is bounded and these are a constant number of checks, all of which run in constant time.

Discussion. DYN heavily relies on having a GJT pair (T, N) to process queries. If, for a CQ Q there exists some GJT pair (T, N) such that $Q \equiv \mathcal{Q}[T, N]$ then Q is said to be *free-connex acyclic*, and (T, N) is called a *GJT pair for* Q. Q is *acyclic* if $full(Q) \equiv \mathcal{Q}[T]$ for some T.[1] Not all CQs are (free-connex) acyclic. For instance, the triangle query $r(x, y) \bowtie s(y, z) \bowtie t(x, z)$ is the prototypical example of a non-acyclic query. Furthermore, $\pi_{x,z}(r(x, y) \bowtie s(y, z))$ is acyclic but not free-connex acyclic. Since every free-connex acyclic CQ is acyclic, this example shows that free-connex acyclic CQs form a strict subclass of the acyclic CQs. Recent analysis of query logs show that free-connex acyclic queries occur very frequently in practice [5]. We refer readers interested in algorithms for computing GJT pairs for GCQs to [14].

One may wonder whether algorithms with the same properties as DYN can be obtained for CQs that are not free-connex acyclic. It is known that this is not possible for the class of all acyclic CQs, unless multiplication of two $n \times n$ binary matrices can be computed in $\mathcal{O}(n^2)$ time [3]. Using further complexity-theoretic assumptions, it is possible to show that this is also not possible for the class of all CQs [6].

It is also known [4] that, unless the Online Matrix-Vector Multiplication conjecture [11] is false, the class of queries that allow both (1) constant-delay enumeration of query results and (2) update processing time $\mathcal{O}(|u|)$ for every update u, is exactly the class of so-called *q-hierarchical queries.* While we forego a formal definition of this class, we show in [12] that a CQ Q is q-hierarchical if, and only if there exists a GJT pair (T, N) for Q such that T is simple. Since DYN has update time $\mathcal{O}(|u|)$ for exactly these queries, DYN hence meets the theoretical lower bound.

For readers familiar with the Yannakakis algorithm [23] it may not be obvious from the description above why DYN can be claimed to be a dynamic version of Yannakakis. We refer to [12] for a discussion.

4 Dealing with θ-joins

To extend DYN to also process θ-joins, it is instructive to consider the GCQ $Q = (R(x, y) \bowtie S(y, z) \mid x < z)$ where the θ-join is an inequality-join. To obtain CDE for Q, assume that we have already computed $Y = \pi_{x,y}(\sigma_{x<z}(R(x, y) \bowtie S(y, z)))$ and that, moreover, we have a more powerful index structure I that allows, for any tuple $\{x, y\}$-tuple $\vec{t}$ over, to enumerate $\sigma_{x<z}(S(y, z) \ltimes \vec{t})$ with constant delay. We can then obviously enumerate Q with constant delay by iterating

[1] There exists many equivalent definitions of when a join query is acyclic, and consequently also of when a CQ with projections is free-connex acyclic. See [12, 14] for a discussion of why the new definition that we give here is equivalent to the existing ones.

over the elements of $\vec{t} \in Y$, and for each such $\vec{t}$, probe I to produce the tuples $\vec{s} \in \sigma_{x<z}(S(y,z) \ltimes \vec{t})$, outputting each $\vec{s} \cup \vec{t}$. Since $\sigma_{x<z}(S(y,z) \ltimes \vec{t})$ allows CDE, the entire procedure is CDE. The key question then, is how we can build this more powerful index structure I. The solution is to build a normal (hash-based) index J of S on y but use sorting to store the index in such a way that for every $\vec{t}$ over y this index returns a pointer to $S \ltimes \vec{t}$ for which tuples are enumerated in descending order on z. This now supports CDE of $\sigma_{x<z}(S(y,z) \ltimes \vec{t})$, for every $\vec{t}$ over $\{x, y\}$: use J to enumerate $R \ltimes \vec{t}$ with constant delay and in decreasing order on z. Yield the current tuple $\vec{s}$ that is being enumerated in this fashion, provided that $\vec{t}(x) < \vec{s}(z)$. As soon as $\vec{t}(x) \geq \vec{s}(z)$ we know that all subsequent $\vec{s}$ will fail the inequality, and we can hence terminate. This example forms the basic intuition in how we can extend Dyn to deal with *GCQs* with inequality joins. We next sketch the generalization to arbitrary θ-joins, and refer to [13] for detailed exposition.

First, in the presence of θ-joins, a GJT pair is defined exactly as in Definition 3.1 except that now additionally every edge $p \to c$ from parent p to child c is labeled by a set $\mathit{pred}(p \to c)$ of predicates. It is required that every predicate $\theta(\overline{z})$ in this set satisfies $\overline{z} \subseteq \mathit{var}(p) \cup \mathit{var}(c)$. The query $\mathcal{Q}[T, N]$ associated to (T, N) then becomes $\pi_{\mathit{var}(N)}(\mathcal{Q}[T] \mid \wedge_{\theta(\overline{z}) \in \mathit{pred}(T)} \theta(\overline{z}))$. Here, $\mathit{pred}(T)$ are all the predicates occurring on edges in T.

Next, (T, N)-reps are extended to account for predicates. Concretely, the inductive definition of ρ_n in the T-reduct becomes:

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

Here $\mathit{pred}(n)$ denotes the set of all predicates on the edges from n to its children in T. The indexes that we need to maintain are modified as follows: P_n should now allow CDE of $\sigma_{\mathit{pred}(p \to n)}(\rho_n \ltimes \vec{t})$, for every $\vec{t}$ over $\mathit{var}(p)$, where p is n's parent. S_n should allow CDE of $\sigma_{\mathit{pred}(p)}(\rho_n \ltimes \vec{t})$ for every tuple $\vec{t}$ over $\mathit{var}(m)$ where m is the sibling of n. The exact design of these indexes of course depends on the semantics of the predicates included in T; for inequality predicates we have sketched above how they work. The generalization of Dyn works as long as we have these indexes.

Finally, Algorithm 1 is modified so that in Lines 6, 8 and 9 we iterate over $\sigma_{\mathit{pred}(n \to c)}(\rho_c \ltimes \vec{t})$ resp. $\sigma_{\mathit{pred}(n \to c_1)}(\rho_{c_1} \ltimes \vec{t})$ and $\sigma_{\mathit{pred}(n \to c_2)}(\rho_{c_2} \ltimes \vec{t})$. Lines 19, 21, 23 are modified to compute the Δ GMRs under the now-modified definition of (T, N)-rep. We refer to the general version of Dyn with arbitrary θ-joins as GDyn, and the version where all θ-joins are inequalities as IEDyn.

Theorem 4.1. *Let (T, N) be a fixed GJT pair. Given a (T, N)-rep of db, both* GDyn *and* IEDyn *correctly enumerate $\mathcal{Q}[T, N](db)$ and update the (T, N)-rep to a rep of $db + u$ under update u. In the case that all predicates in θ-joins are inequalities,* IEDyn *has the following complexity. If there is at most one inequality on each edge in T, then* IEDyn *enumerates with constant delay and has $\mathcal{O}(M \cdot \log(M))$ update time where $M = (|db| + |u|)$. If T has some edge that contains multiple inequalities, the delay is $\mathcal{O}(\log(|db|))$ and the update time is $\mathcal{O}(M^2 \cdot \log(M))$.*[2]

5 Experimental Evaluation

We have implemented (IE)Dyn as a query compiler that generates executable code in the Scala programming language. The generated code instantiates a (T, N)-rep for a query Q and defines *trigger functions* that are used for maintaining the (T, N)-rep under updates.

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

It should be noted that our implementation also supports the processing of general acyclic GCQs that are not necessarily free-connex. This is done using the following simple strategy. Let Q be acyclic but not free-connex. First, compute a free-connex acyclic approximation Q_F of Q. Q_F can always be obtained from Q by extending the set of output variables of Q. In the worst case, we need to add all variables, and Q_F becomes the full join underlying Q. Then, use (IE)Dyn to maintain a (T, N)-rep for Q_F. When operating in push-based mode, for each update u, we use the (T, N)-rep to delta-enumerate $\Delta Q_F(db, u)$ and project each resulting tuple to materialize $\Delta Q(db, u)$ in an array. Subsequently, we copy this array to the output. Note that the materialization of $\Delta Q(db, u)$ here is necessary since the delta enumeration can produce duplicate tuples after projection. When operating in pull-based mode, we materialize $Q(db)$ in an array, and use delta enumeration of Q_F to maintain the array under updates. Of course, under this strategy, we require $\Omega(|Q(db)|)$ space in the worst case, just like (H)IVM would, but we avoid the (partial) materialization of delta queries. Note the distinction between the two modes: in push-based mode $\Delta Q(db, u)$ is materialized (and discarded once the output is generated), while in pull-based mode $Q(db)$ is materialized upon requests. Finally, our implementation also supports common aggregates like `SUM` and `AVG`, see [12] for more information.

5.1 Conjunctive Queries

We evaluate a subset of queries available in the industry-standard benchmarks TPC-H and TPC-DS. In particular, we evaluate those queries involving only equijoins, whose FROM-WHERE clauses 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. We omit the ORDER BY and LIMIT clauses, we replaced the

[2] In [13] there was an incorrect claim: we stated that updates could be processed in time $\mathcal{O}(M \cdot log(M))$ in this last case. We then found a bug in our algorithm and we currently do not know if this bound can be achieved. See [14] for a proof that IEDyn runs in the bounds claimed here.

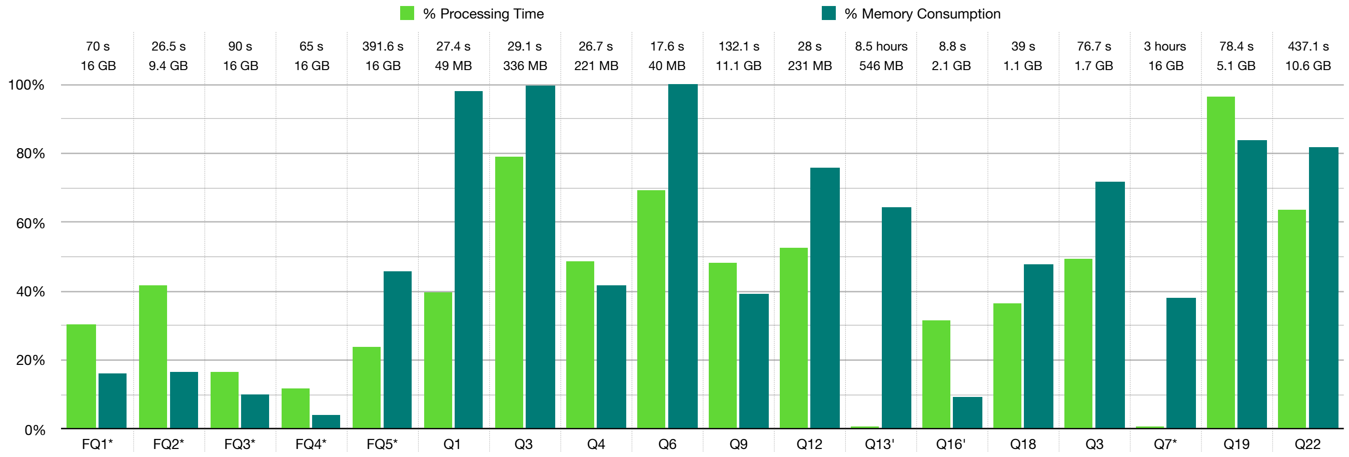

Figure 5: DYN usage of resources as a percentage of the resources consumed by DBToaster (lower is better).

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: CQ benchmark stream sizes.

left-outer join in TPC-H query Q13 by an equijoin, and modified TPC-H Q16 to remove an inequality. See [12] for the full query specification.

Our workload consist of a stream of updates, where each update consists of a single-tuple insertion. The streams were generated using the TPC-H and TPC-DS data generators. The number of tuples in each stream is depicted in Table 1.

We compare IEDYN with DBToaster [16] using *memory footprint* and *update processing time* as comparison metrics. DBToaster is a state-of-the-art implementation of HIVM. It operates in pull-based mode, and is optimized for aggregations over equi-joins. DBToaster has been extensively tested for such queries and has proven to be more efficient than a commercial database management system, a commercial stream processing system and an IVM implementation [16]. It is therefore an interesting implementation to compare to. DBToaster compiles given SQL statements into executable trigger programs in different programming languages. We compare against those generated in Scala from the DBToaster Release 2.2.[3]

Figure 5 depicts the resources used by DYN as a percentage of the resources used by DBToaster, both operating in pull-based mode. For each query, we plot the percentage of memory used by DYN considering that 100% is 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 5. Note that DYN never runs out of memory, and times reported for DYN are the times required to process the entire update stream.

From Figure 5 we see that for full join queries (FQ1-FQ5), DYN outperforms DBToaster by close to one order of magnitude in both memory consumption and processing time, illustrating the effectiveness of maintaining (T, N)-reps rather than the query results themselves, especially when these results are large. For aggregate queries, Figure 5 shows that DYN can significantly improve the memory consumption of HIVM while improving processing time—up to two orders of magnitude for TPC-H Q13’ and TPC-DS Q7. See [12] for an in-depth discussion.

While the T-reps maintained by IEDYN feature constant delay enumeration, this theoretical notion hides a constant factor that could decrease performance in practice when compared to full materialization. Experiments detailed in [12] show that this not the case: DYN’s enumeration time is competitive with DBToaster.

5.2 Conjunctive Queries with Inequalities

To gauge the effectiveness of IEDYN on GCQs that feature inequality joins, we evaluate the acyclic queries listed in Table 2 on synthetically-generated streams of single-tuple insertion updates. The sizes of the update streams are intentionally kept low, since they generate huge output sizes (cf. Table 2).

Here we only compare IEDYN with Esper[4] but refer to [13] for a more detailed comparison against other state of the art

[3] `https://dbtoaster.github.io/`

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

Query	Expression	Join	Type	|Stream|	|Output|
GCQ_1	$R(a,b,c) \bowtie S(d,e,f) \mid a < d$	$<$	Full	12k	$18,017$k
GCQ_2	$R(a,b,c) \bowtie S(d,e,f) \bowtie T(g,h,i) \mid a < d \wedge e < g$	$<$	Full	2.7k	$178,847$k
GCQ_3	$R(a,b,c) \bowtie S(d,e,f) \bowtie T(g,h,i) \mid a < d \wedge d < g$	$<$	Full	2.7k	$90,425$k
GCQ_4	$R(a,b,c) \bowtie S(d,e,f,k) \bowtie T(g,h,i,k) \mid a < d \wedge d < g$	$<,=$	Full	21k	$297,873$k
GCQ_5	$\pi_{a,b,d,e,f,g,h}(\text{GCQ}_3)$	$<$	Free-connex	2.7k	$114,561$k
GCQ_6	$\pi_{d,e,f,g,h,k}(\text{GCQ}_4)$	$<,=$	Free-connex	21k	$99,043$k
GCQ_7	$\pi_{b,c,e,f,h,i}(\text{GCQ}_3)$	$<$	Free-connex	2.7k	$114,561$k
GCQ_8	$\pi_{b,c,e,f,h,i}(\text{GCQ}_4)$	$<,=$	Free-connex	21k	$297,873$k

Table 2: GCQ benchmark queries, together with update stream and result sizes, $k = 1000$.

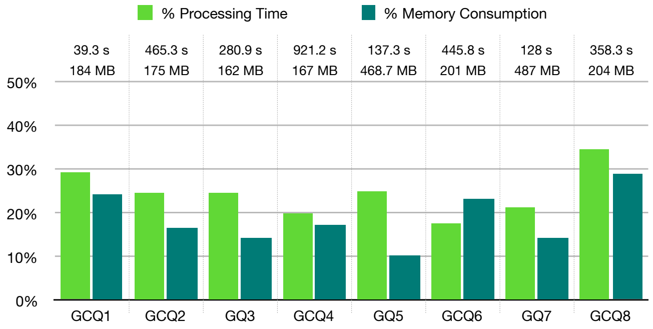

Figure 6: IEDyn resource usage as a percentage of the resources used by Esper (lower is better).

systems. Esper is a complex event processing engine with a relational model based on Stanford STREAM [2]. It operates in push-based mode. We use the Java-based open source implementation[4] for our comparisons.

Figure 6 depicts the resources used by IEDyn as a percentage of the resources used by Esper, both operating in push-based mode. IEDyn significantly outperforms Esper on all full join queries (GCQ_1–GCQ_4). We note that for these queries, even in push-based mode IEDyn can support the enumeration of query results from its data structures at any time while competing push-based systems have no such support. Hence, IEDyn is not only more efficient but also provides more functionality. IEDyn also significantly outperforms Esper on free-connex queries GCQ_5 and GCQ_6 with more than a threefold improvement in processing time and an order of magnitude improvement in memory usage on Q_7. For non-free-connex queries GCQ_7 and GCQ_8, IEDyn continues to significantly outperform Esper in processing time, showing an order of magnitude improvement in memory usage for GCQ_7.

6 Summary

Traditional techniques for dynamic query evaluation are based either on materialization (to avoid recomputation of subresults), or on recomputation of (to avoid the space overhead of materialization). We have shown that both techniques are suboptimal: instead of materializing subresults, one can use Dynamic Yannakakis to maintain a data structure that is succinct; and yet supports all operations one commonly expects from materialization: enumeration with constant delay as well as fetching single tuples in constant time. Our experiments against state-of-the art engines in different domains show that this can improve performance by orders of magnitude.

7 References

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

[2] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: the stanford data stream management system. In *Data Stream Management - Processing High-Speed Data Streams*, pages 317–336. Springer, 2016.

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

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

[5] A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. Full version of PVLDB 11(2) paper, under submission. Obtained through personal communication.

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

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

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

[9] 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.

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

[11] 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.

[12] M. Idris, M. Ugarte, and S. Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proc. of SIGMOD*, pages 1259–1274, 2017.

[13] M. Idris, M. Ugarte, S. Vansummeren, H. Voigt, and W. Lehner. Conjunctive queries with inequalities under updates. *PVLDB*, 11(7):733–745, 2018.

[14] M. Idris, M. Ugarte, S. Vansummeren, H. Voigt, and W. Lehner. Conjunctive queries with θ-joins under updates. Technical report, 2019. Available at http://arxiv.org/abs/1905.09848.

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

[16] 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.

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

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

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

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

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

[22] W. Wang, J. Gao, M. Zhang, S. Wang, G. Chen, T. K. Ng, B. C. Ooi, J. Shao, and M. Reyad. Rafiki: Machine learning as an analytics service system. *PVLDB*, 12(2):128–140, 2018.

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