On the difference between Complex Event Processing and Dynamic Query Evaluation

Martín Ugarte and Stijn Vansummeren

Université Libre de Bruxelles

{martin.ugarte, stijn.vansummeren}@ulb.ac.be

Abstract. The increasing number of applications that require processing and analyzing high-throughput information in real time has fostered a new field of study known as Complex Event Processing (CEP). The claimed objective of CEP is to develop techniques that are able to cope with the high-throughput requirements of modern Big Data applications. Also, it is commonly argued that CEP systems are different from relational reactive systems such as Active Database Management Systems (ADBMSs) or Data Stream Management Systems (DSMSs) because the latter see new data elements as database transactions (generally insertions or deletions) whose order is not relevant. On the contrary, CEP systems see new data elements as events (e.g. sensor measurements) whose arrival time and order have a semantic meaning. Unfortunately, these differences come from a high-level description of the underlying applications, but do not reveal fundamental computational differences between the requirements of CEP systems and relational systems. Moreover, recent developments in Dynamic Query Evaluation (DQE) show that general techniques in the relational setting can be more efficient than current CEP algorithms.

In this paper, we study whether there is a fundamental difference between the computational requirements of CEP and DQE. To answer this question, we identify two concrete assumptions of CEP and investigate their effects in terms of evaluation complexity. Concretely, we show a realistic CEP query that, if the *Online Matrix-vector multiplication* (OMv) conjecture holds, cannot be evaluated with sub-linear time per tuple followed by sub-linear-delay enumeration, but under the CEP assumptions can be evaluated with constant time per tuple followed by constant-delay enumeration. Sub-linear here means $O(n^{1-\varepsilon})$, where n is the size of the active domain.

1 Introduction

In recent years, analyzing high-throughput streams in real time has become a crucial task for many communities. Examples range from Stream Processing [24] and Financial Systems [9] to Industrial Control Systems [1] and On-line Machine Learning [23]. The diversity of these communities and the nature of their applications has resulted in a variety of frameworks and systems to analyze data

streams. These systems fall in different categories (see [11] for an extensive survey), out of which Complex Event Processing and Dynamic Query Evaluation are two prominent examples.

The term Dynamic Query Evaluation (DQE) refers to techniques whose aim is to efficiently maintain an up-to-date version of a query result under updates to an underlying database. DQE is usually based on a form of Incremental View Maintenance (IVM) [3, 6, 7, 18], whose objective is to maintain a materialized version of query results under updates by applying relational operations over cached results and sub-results. In particular, in DQE the input is a stream of database transactions that can insert as well as delete tuples. Systems based on DQE find their origins in Active Database Management Systems (ADBMS) and Data Stream Management Systems (DSMS), that were designed to execute relational operations every time a transaction was executed in a database [2].

Complex Event Processing (CEP) refers to techniques where the objective is to detect certain *temporal patterns* in a high-throughput data stream. Therefore, elements in the input stream are not seen as database transactions but as *events*, that represent real-world observations and that thus cannot be retracted. Moreover, the order at which these events arrive to the system has a semantic meaning; CEP patterns commonly express sequences based on this temporal order. CEP and DQE are usually considered very different as is perhaps best illustrated by Cugola and Margara who, in their well-known survey [11], argue that *"The concepts of timeliness and flow processing are crucial for justifying the need for a new class of systems"*. In line with this difference, specialized CEP systems were developed to deal with the throughput requirements of applications scenarios involving CEP patterns. Prominent examples of CEP systems are SASE [26], Cayuga [12], EsperTech [13], TESLA/T-Rex [10], RTEC [4], among others. While CEP and DQE have hence historically been considered different in the literature, this perceived difference is not based on fundamental computational properties, but rather on a high-level description of the targeted applications of CEP and DQE. Several recent trends make the perceived difference between CEP and DQE rather shallow. First of all, CEP and DQE application scenarios seem to intermingle. For this reason, modern distributed stream processing systems (such as Apache Flink[1], Spark Streaming[2] or Apache Storm[3]) mingle both incremental query maintenance capabilities and CEP features like time windows or time-based semantics. Second, DQE has received new attention in the last couple of years [15,21] based on a simple yet important idea: instead of maintaining a materialized version of the query results, maintain a data structure that can be updated efficiently under changes to the underlying database, and from which the query results can be efficiently generated. Recent work shows that relational techniques based on this idea can be more efficient than the CEP systems cited above [16].

[1] https://flink.apache.org
[2] https://spark.apache.org/streaming/
[3] https://storm.apache.org

This hence raises the question: *is there a fundamental computational difference between CEP and DQE*? In order to answer this question, we take a fundamental viewpoint, and define CEP systems as systems in which the input is a stream of tuples. In line with the fact that events cannot be retracted, the stream in insertion-only. Moreover, in line with the fact that arrival order is important, we require that every tuple contains a special attribute that represents the arrival time to the system. The value of this attribute is globally increasing across the stream. In contrast, DQE systems take as input a stream of both insertions and deletions with no further assumption on the attributes.

Concretely, we answer the question by showing a query that can be answered efficiently under the CEP assumptions, but not under the DQE assumptions. The notion of efficiency here corresponds to process each new event in constant time (under data complexity) and be able to output the results at any moment with constant-delay enumeration.

2 Preliminaries

Tuples, databases, predicates and streams. We assume the existence of three disjoint infinite sets: A domain $\mathbf{D}$, a set of variables $\{x, y, z, \ldots\}$ (which play the role of attribute names), and a set of relation names $\{r, s, t, \ldots\}$.

An *atom* is an expression of the form $r(\overline{x})$, where r is a relation name and $\overline{x}$ is a finite set of variables. A tuple over $r(\overline{x})$ (or simply over $\overline{x}$) is a function $\boldsymbol{t} : \overline{x} \to \mathbf{D}$. A relation R over $r(\overline{x})$ is a finite set of tuples over $r(\overline{x})$. A schema $\mathcal{S}$ is a finite set of atoms with different relation names, and a database D over schema $\mathcal{S}$ is a set of relations, one over each atom of $\mathcal{S}$. A stream over schema $\mathcal{S}$ is a possibly infinite sequence $S = (S[1], S[2], \ldots)$, where each $S[i]$ is a pair $(\boldsymbol{t}_i, \circ_i)$ consisting of a tuple $\boldsymbol{t}_i$ over any of the atoms in $\mathcal{S}$ and a symbol $\circ_i \in \{+, -\}$ that represents whether $S[i]$ is an insertion or a deletion. A stream is called insertion-only if all of its elements are insertions. We might abuse notation and assume the elements of an insertion-only stream are tuples. Given a stream S, the sub-stream $(S[i], \ldots, S[j])$ is denoted by $S_{i,j}$, and the database generated by S up to position i, denoted by $S[1..i]$, is the database consisting of all tuples $\boldsymbol{t}$ that occur in S up to position i (inclusive), and whose last occurrence in $S_{1,i}$ is an insertion (i.e. $(\boldsymbol{t}, +)$).

Finally, a predicate over $\overline{x}$ is a (not necessarily finite) decidable set P of tuples over $\overline{x}$. For example, if $\overline{x} = \{x, y\}$ then $P(\overline{x}) = \{(x, y) \mid x < y\}$ is the predicate of all tuples $(x_0, y_0) \in \mathbf{D}^2$ satisfying $x_0 < y_0$. As usual, we use shorthand notation for predicates (e.g. $P(\overline{x}) = x < y$) and write $P(\boldsymbol{t})$ to indicate that $\boldsymbol{t} \in P$.

Query Language. We follow the query language of Generalized Conjunctive Queries (GCQs) introduced in [16].

Definition 1. *A GCQ is an expression of the form*

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

where $r_1(\overline{x_1}), \ldots, r_n(\overline{x_n})$ *are atoms,* $P_1, \ldots, P_m$ *are predicates over* $\overline{z_1}, \ldots, \overline{z_m}$*, respectively, and both* $\overline{y}$ *and* $\bigcup_{i=1}^{m} \overline{z_i}$ *are subsets of* $\bigcup_{i=1}^{n} \overline{x_i}$*.*

Given a database D and a GCQ Q of the form (1), the evaluation of Q over D is denoted $Q(D)$ and is performed in the expected way: first compute the natural join $r_1(\overline{x_1}) \bowtie \cdots \bowtie r_n(\overline{x_n})$, from the result keep only those tuples that satisfy all predicates $P_1, \ldots, P_m$, and finally project them over $\overline{y}$. We illustrate the semantics of GCQs with an example, for a full definition see [16].

Example 1. Consider the following GCQ.

$$\pi_{x,y,ts_3} \left(r(x, ts_1) \bowtie s(x, y, ts_2) \bowtie t(y, ts_3) \mid ts_1 < ts_2 < ts_3 \wedge (ts_3 - ts_1) < 5 \right) \quad (2)$$

Intuitively, the query specifies that we should take the natural join between $r(x, ts_1)$, $s(x, y, ts_2)$ and $t(y, ts_3)$, from this result keep only those tuples that satisfy $ts_1 < ts_2 < ts_3$ and $(ts_3 - ts_1) < 5$, and finally project over $\{x, y, ts_3\}$.

From the definitions of databases, streams and the semantics of GCQs it can be seen that we use set semantics. We stress, however, that this is only for simplicity, and the results in this paper can be extended easily to bag semantics.

GCQs for Complex Event Processing. It has been claimed in the past that CEP languages are generally informal and non-standard [10,11,26], presenting a variety of primitives and operators with sometimes problematic semantics. Nevertheless, GCQs capture some fundamental requirements of CEP. For example, in CEP frameworks users can specify that events must occur in a particular order and inside a bounded time window. Although this is not directly provided by built-in features of GCQs, it can be simulated by adding a globally-increasing attribute to tuples and using inequality predicates. For example, in query (2) we could interpret ts_1, ts_2 and ts_3 as the time of arrival of each event. Then, this query would select three events of type r, s and t that arrived in that same order and inside a time window of 5 units of time (we assume seconds).

In contrast to GCQs, CEP languages usually have sequencing and time windows as primitives. For example, under the assumption that ts_1, ts_2 and ts_3 represent arrival order, query (2) can be written equivalently in SASE [26] as

```
SEQ(r,s,t) WHERE r.1 = s.1 AND s.2 = t.1 WITHIN 5 seconds.
```

The `SEQ` operator indicates that the events must arrive in the expected order and the `WITHIN` clause that they have to occur within a time window of 5 seconds, avoiding the explicit inequalities over the timestamps. The `WHERE` clause indicates that the first attribute of r must be equal to the first attribute of s and the second attribute of s must be equal to the first attribute of t, which is equivalent to using variable names in GCQs.

For the purposes of this paper, we only require the CEP features of sequencing, time windows and filtering; it is not relevant that other features of CEP languages like iteration (Kleene closure) are not expressible by means of GCQs.

3 Evaluating Queries over Dynamic Databases

Since we are interested in evaluating GCQs over streams, we need to formally define what evaluation means. Traditionally, the objective of DQE algorithms like IVM has been to maintain the *current* version of the result materialized at all times. Therefore, the processing of an update includes the corresponding modification of the materialized result. In contrast, recent DQE algorithms do not maintain the full materialized output, but a data structure from which the result can be generated [5]. The idea behind this is to separate the enumeration of results from the process of updating the data structure when the underlying database changes. Note that this implies that the time required to react to an update can be much shorter than the size of the change to the output, because the data structure can be a *compressed representation* of the output. This is actually the main reason behind the efficiency gain in recent DQE algorithms [15,16,19].

When evaluating a query Q over a stream S, the elements of S are consumed one by one in order. Once the i^{th} element of S is processed, we expect to be able to generate $Q(S[1..i])$ (recall that $S[1..i]$ is the database *produced* by S up to position i). Following this, a DQE algorithm $\mathcal{A}$ for Q consists of a data structure $D_{\mathcal{A}}$ and two routines $\texttt{PROCESS}_{\mathcal{A}}$ and $\texttt{ENUM}_{\mathcal{A}}$ such that for every stream S:

- when $\texttt{PROCESS}_{\mathcal{A}}$ receives an element of the stream, it can modify $D_{\mathcal{A}}$ and;
- after $\texttt{PROCESS}_{\mathcal{A}}$ has been called with the first i elements of the stream, $\texttt{ENUM}_{\mathcal{A}}$ enumerates $Q(S[1..i])$ from $D_{\mathcal{A}}$.

Complexity. We consider query evaluation in main memory and measure time under data complexity [25]. That is, the query is considered to be fixed and not part of the input. This makes sense under DQE, where the query is known in advance and the data is constantly changing. In particular, the number of atoms and predicates in the query, their arity, and the length of the query are all constant. Also, note that we need to measure result enumeration and update processing separately. Update processing is simply measured as the worst-case time required to process an element from the stream, i.e. the worst-case time required by $\texttt{PROCESS}_{\mathcal{A}}$. Formally, given a function f from streams to natural numbers, we say that a DQE algorithm $\mathcal{A}$ provides f update-time if, for every stream S and $i \in \mathbb{N}$, $\texttt{PROCESS}_{\mathcal{A}}$ takes time $O(f(S_{1,i}))$ to process $S[i]$.

The efficiency of enumeration is slightly more involved, since it is measured as the delay that the enumeration procedure takes between generating two consecutive results [5,22]. This is defined as follows.

Definition 2. *A routine* ENUM *with access to a data structure D supports* enumeration *of a set E if* ENUM(D) *outputs each element of E exactly once. Moreover,* ENUM(D) *enumerates E with delay $d \in \mathbb{N}$, 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 bounded by d.*

Given a function f from streams to natural numbers, we say that a DQE algorithm $\mathcal{A}$ for query Q provides f-delay enumeration if for every stream S and every $i \in \mathbb{N}$, after $\texttt{PROCESS}_{\mathcal{A}}$ has processed the first i elements of S, $\texttt{ENUM}_{\mathcal{A}}$ enumerates $Q(S[1..i])$ with delay $O(f(S_{1,i}))$. If f is a constant function, $\mathcal{A}$ is said to provide constant-delay enumeration (CDE) of $Q(S[1..i])$.

Computational Model. Another important consideration when studying the efficiency of DQE algorithms is the computational model. Indeed, when dealing with fine-grained notions like constant-time per update or CDE, the computational model might affect the complexity.

We follow the extended RAM model from [15]. Concretely, we assume a model of computation where domain values take $O(1)$ space and memory lookups are $O(1)$. We further assume that every relation R can be represented by a data structure that allows (1) enumeration of all tuples in R with constant delay (i.e. the existence of a routine as defined above); (2) lookups in $O(1)$, meaning that given $\boldsymbol{t}$ we can decide in constant time whether $\boldsymbol{t} \in R$; (3) single-tuple insertions and deletions in $O(1)$ time; while (4) having size that is proportional to the number of tuples in R.

We further assume the existence of linear-size hash-maps. Concretely a hash map H over a set of variables $\overline{x}$ can store an arbitrary data structure $H(\boldsymbol{t})$ for each tuple $\boldsymbol{t}$ over $\overline{x}$, using size proportional to the sum of the sizes of the data structures stored. Moreover, given a tuple $\boldsymbol{t}$ we can access $H(\boldsymbol{t})$ in constant time. These assumptions amount to perfect hashing of linear size [8]. Although this is not realistic for practical computers [20], it is well known that complexity results for this model can be translated, through amortized analysis, to average complexity in real-life implementations [8].

4 A Computational Difference

As mentioned in the introduction, the differences between DQE and CEP generally come from high-level descriptions of the targeted applications, and not from fundamental differences in their processing techniques. It makes sense then to discuss whether CEP really requires new computational insights for achieving better performance, or DQE algorithms are already optimal for the requirements of CEP. To understand this, we first need to identify the concrete requirements and restrictions that distinguish CEP from DQE. As discussed in the introduction, CEP adds to the dynamic evaluation problem (1) the requirement of selecting tuples based on their arrival times and (2) the restriction that witnessed tuples correspond to real-world events and thus cannot be retracted. This can be concretely reflected in GCQ evaluation by

1. interpreting some attributes as timestamps and assuming that they occur in a globally increasing manner, and
2. assuming that streams are insertion-only.

We show that taking these assumptions into account can make a difference in terms of complexity of DQE algorithms. Concretely, we prove that the GCQ

$$Q = \pi_{y,ts_3}(r(x, ts_1) \bowtie s(x, y, ts_2) \bowtie t(y, ts_3) \mid ts_1 < ts_2 < ts_3 \wedge (ts_3 - ts_1) < 5)$$

can be evaluated with constant update time followed by CDE when restricted to streams that satisfy the two assumptions above, but not in general.

We start by presenting the hardness result. We show that there is no DQE algorithm that evaluates Q over streams with sub-linear update time followed by sub-linear-delay enumeration. Here, sub-linear means $O(n^{1-\varepsilon})$, where n is the number of elements in the active domain of the database produced by the processed portion of the stream. We use the dichotomy proved recently by Berkholz et. al. in [5], which is based on the *Online Matrix-vector Multiplication* (OMv) conjecture (see [14]). The full description of the dichotomy is rather involved, but it implies that a if a CQ is not *q-hierarchical* then it cannot be evaluated over dynamic datasets with sub-linear time per update followed by sub-linear-delay enumeration. For the sake of space, we do not give here the formal definition of q-hierarchical queries, but only state that the query $Q' = \pi_y(u(x) \bowtie v(x, y) \bowtie w(y))$ is not q-hierarchical.

Lemma 1. *If the OMv conjecture holds, there is no DQE algorithm for*

$$Q = \pi_{y,ts_3}(r(x, ts_1) \bowtie s(x, y, ts_2) \bowtie t(y, ts_3) \mid ts_1 < ts_2 < ts_3 \wedge (ts_3 - ts_1) < 5)$$

with sub-linear time per update followed by sub-linear-delay enumeration.

Proof (Sketch). We prove this by contradiction. Assume that there is a DQE algorithm $\mathcal{A}$ that evaluates Q with sub-linear update time and sub-linear-delay enumeration. We use this algorithm to define a second algorithm $\mathcal{A}'$ that evaluates query $Q' = \pi_y(u(x) \bowtie v(x, y) \bowtie w(y))$ with the same bounds. First, the data structure $D_{\mathcal{A}'}$ is exactly the same as the data structure $D_{\mathcal{A}}$. Upon the arrival of an element $(\boldsymbol{t}, \circ)$, the routine $\texttt{PROCESS}_{\mathcal{A}'}$ will simply call $\texttt{PROCESS}_{\mathcal{A}}(\boldsymbol{u}, \circ)$, where $\boldsymbol{u}$ is defined as follows: if $\boldsymbol{t} = u(x_0)$, then $\boldsymbol{u} = r(x_0, 1)$; if $\boldsymbol{t} = v(x_0, y_0)$, then $\boldsymbol{u} = s(x_0, y_0, 2)$; if $\boldsymbol{t} = w(y_0)$, then $\boldsymbol{u} = t(y_0, 3)$. Since $\texttt{PROCESS}_{\mathcal{A}}(\boldsymbol{u}, \circ)$ takes sub-linear time, it is clear that $\texttt{PROCESS}_{\mathcal{A}'}(\boldsymbol{t}, \circ)$ also takes sub-linear time. Finally, the routine $\texttt{ENUM}_{\mathcal{A}'}$ is equivalent to $\texttt{ENUM}_{\mathcal{A}}$ except for the fact that it needs to project out ts_3. It is clear that this process will generate the expected output with sub-linear-delay enumeration: the filters will be satisfied because ts_1 is always 1, ts_2 is always 2 and ts_3 is always 3. Moreover, enumeration is correct and without duplicates (even though we project away ts_3) exactly because ts_3 is always 3. We hence have our desired contradiction as we have constructed a DQE algorithm with sub-linear update time followed by sub-linear-delay enumeration that evaluates Q', which is not q-hierarchical. □

Next, we show a DQE algorithm $\mathcal{A}$ that evaluates Q with constant time per update followed by CDE under the assumptions that streams are insertion-only and ts_1, ts_2 and ts_3 arrive in a globally increasing fashion. We start by presenting the data structure $D_{\mathcal{A}}$, to then describe how $\texttt{PROCESS}_{\mathcal{A}}$ maintains it. After the first i elements of S are processed, $D_{\mathcal{A}}$ contains three elements.

Algorithm 1 $\texttt{UPDATE}_{\mathcal{A}}$: Updates $D_{\mathcal{A}}$ upon receiving a tuple $\boldsymbol{t}$.

```
1: Input: t.
2: if t = r(x, ts_1) then
3:     H_R(x) ← ts_1.
4: else if t = s(x, y, ts_2) then
5:     if x ∈ H_R then
6:         H_S(y) ← H_R(x).
7: else if t = t(y, ts_3) then
8:     if y ∈ H_S and ts_3 − H_S(y) < 5 then
9:         Insert (y, ts_3) into T
```

- A hash-map H_R that maps each x to the maximum ts_1 for which $r(x, ts_1) \in S_{1,i}$. Intuitively, $H_R(x)$ is the last time that x occurred in an r tuple.
- A hash-map H_S mapping each y to a timestamp. The timestamp $H_S(y)$ is the largest ts_1 for which there exist tuples $r(x_0, ts_1)$ and $s(x_0, y, ts_2)$ in $S_{1,i}$ satisfying $ts_1 < ts_2$.
- A relation T over $\{y, ts_3\}$ that contains the actual output $Q(S[1..i])$.

Having access to $D_{\mathcal{A}}$ the enumeration procedure is trivial, since the output is already materialized in relation T of $D_{\mathcal{A}}$. We proceed then to describe the update process.

Update processing. The objective of $\texttt{PROCESS}_{\mathcal{A}}$, shown in Algorithm 1, is to maintain all elements of $D_{\mathcal{A}}$ consistent with the definitions given above. Whenever a new tuple $\boldsymbol{t}$ of type $r(x, ts_1)$ arrives, it suffices to insert ts_1 as the last timestamp of x (Line 3). Note that we can only do this because we assume that timestamps arrive in a globally increasing fashion. As such, the value of ts_1 is the maximum across all timestamps seen so far. When a tuple $\boldsymbol{t} = s(x, y, ts_2)$ arrives, we need to check if the x value has already occurred in an r tuple (Line 5), and if so, update the value of $H_S(y)$ to $H_R(x)$. Again, this maintains the consistency because we know that ts_2 is larger than $H_R(x)$. Finally, when a new tuple $\boldsymbol{t}$ of type $t(y, ts_3)$ arrives, we check if we need to insert it into T. To this end, we only need to check the value $H_S(y)$, as this value tells us if there are matching tuples of type r and s in the required time window (Line 8).

It is clear that $\texttt{PROCESS}_{\mathcal{A}}$ runs in constant time under our computational model. Also, because we assume that ts_1, ts_2 and ts_3 are globally increasing, it is easy to see that $\texttt{PROCESS}_{\mathcal{A}}$ correctly maintains the elements of $D_{\mathcal{A}}$.

Theorem 1. *There exists a DQE algorithm that evaluates Q with constant time per tuple and CDE under the assumption that streams are insertion-only and that attributes ts_1, ts_2 and ts_3 are globally increasing.*

In this section we have shown a computational difference between CEP and DQE, giving a concrete motivation for studying techniques that differ from those used in the relational setting. We conclude by further discussing this difference and providing insight into the particular aspects that might make a query *easy* to compute in a CEP setting.

5 Discussion

We have presented the difference between CEP and DQE by means of a very particular query. We decided to present this query because (1) it features both sequencing and time windows, making it realistic for CEP (2) the algorithm to evaluate it with constant time per update followed by CDE allows for a clear presentation. Nevertheless, one possible criticism is that this query is too simple, because every new event can generate at most one output tuple, and therefore constant-delay enumeration is easy to achieve by materializing the output (which is actually what we do). This is a very valid argument; queries for which a single tuple can generate multiple outputs are indeed harder to evaluate. One possibility to present such a query would have been to add variable x to the output of Q (as presented in Query (2) of Example 1). In this case, we can still improve the evaluation performance by using *dynamic fractional cascading* techniques [17], which provide logarithmic update-time followed by logarithmic-delay enumeration. This is still sub-linear in the active domain and therefore an improvement over the general case, but the presentation of this result would have been far more involved. We are not aware of constant-time per update followed by CDE in this case. It is also important to notice that we could have presented a simpler query by removing one of the CEP features. For example, if we remove from Q the time window ($ts_3 - ts_1 < 5$), we obtain a simpler DQE algorithm with the same performance guarantees.

Apart from structural properties of the query, we can also look at the CEP assumptions. At the moment, we do not know whether both assumptions (insertion-only streams and globally-sorted timestamps) are required, nor how is complexity affected if we remove each of them separately. We envision this and a general understanding of the query properties that affect complexity as future work.

Finally, in CEP it is common to distinguish between push-based and pull-based systems (see for example [11, 16]). Intuitively, in a pull-based setting the system must be ready to generate the complete answer $Q(S[1..i])$ whenever a user asks for it. This corresponds to the semantics presented in this paper. On the contrary, when element $S[i]$ arrives to a push-based system, the system must inform the user of the output's *delta*, i.e. the difference between $Q(S[1..i])$ and $Q(s[1..i-1])$. Interestingly, for the case of push-based systems we can evaluate Query (2) of Example 1 with logarithmic update time followed by constant-delay enumeration (of the output's delta). Moreover, if we take out the time window filter but keep x in the output, we can provide constant-time per update followed by CDE in the push-based case, but we are not aware of an algorithm achieving these times in a pull-based setting. All of these insights suggest that push-based evaluation is simpler than pull-based evaluation; we also envision an in-depth study of this difference as future work.

References

1. *Automation, Production Systems, and Computer-Integrated Manufacturing.* Prentice Hall PTR, 2nd edition, 2000.

2. Corporate Act-Net Consortium. The active database management system manifesto: A rulebase of adbms features. *SIGMOD Rec.*, 1996.
3. Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *VLDB*, 2012.
4. Alexander Artikis, Marek J. Sergot, and Georgios Paliouras. An event calculus for event recognition. *IEEE Transactions on Knowledge and Data Engineering*, 2015.
5. Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *PODS*, 2017.
6. Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.
7. Rada Chirkova and Jun Yang. *Materialized Views*. Now Publishers Inc., 2012.
8. T.H. Cormen. *Introduction to Algorithms, 3rd Edition:*. MIT Press, 2009.
9. Graham Cormode, S Muthukrishnan, and Wei Zhuang. Conquering the divide: Continuous clustering of distributed data streams. In *ICDE*, 2007.
10. Gianpaolo Cugola and Alessandro Margara. Tesla: a formally defined event specification language. In *DEBS*, 2010.
11. Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM CSUR*, 2012.
12. Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *EDBT*, 2006.
13. EsperTech. Esper complex event processing engine. http://www.espertech.com/.
14. Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, 2015.
15. Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *SIGMOD*, 2017.
16. Muhammad Idris, Martin Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. Conjunctive queries with inequalities under updates. In *VLDB*, 2018. To appear.
17. Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 1990.
18. Milos Nikolic, Mohammad Dashti, and Christoph Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *SIGMOD*, 2016.
19. Milos Nikolic and Dan Olteanu. Incremental view maintenance with triple lock factorisation benefits. In *SIGMOD 2018*, 2018. To appear.
20. Christos H. Papadimitriou. Computational complexity. In *Encyclopedia of Computer Science*, pages 260–265. 2003.
21. Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD*, 2016.
22. Luc Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Rec.*, 2015.
23. Shai Shalev-Shwartz. Online learning and online convex optimization. *Found. Trends Mach. Learn.*, (2):107–194, 2012.
24. Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 2005.
25. Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proc. of STOC*, 1982.
26. Haopeng Zhang, Yanlei Diao, and Neil Immerman. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD*, 2014.