Distributed Stream Consistency Checking

Shen Gao[1], Daniele Dell'Aglio[1], Jeff Z. Pan[2], and Abraham Bernstein[1]

[1] IFI, University of Zurich
[shengao,dellaglio,bernstein]@ifi.uzh.ch
[2] The University of Aberdeen
jeff.z.pan@abdn.ac.uk

Abstract. Dealing with noisy data is one of the big issues in stream processing. While noise has been widely studied in settings where streams have simple schemas, e.g. time series, few solutions focused on streams characterized by complex data structures. This paper studies how to check consistency over large amounts of complex streams. The methods we propose exploit reasoning to assess if portions of the streams are compliant to a reference conceptual model. To achieve scalability, our methods runs on state-of-the-art distributed stream processing platforms, e.g. Apache Storm or Twitter Heron. Our first method computes the closure of Negative Inclusions (NIs) for an ontology and registers the NIs as queries. The second method compiles the ontology into a processing pipeline to evenly distribute the workload. Experiments compares the two methods and show that the second one improves the throughput up to 139% with the LUBM ontology and 330% with the NPD ontology.

1 Introduction

Web data streams are changing our lives. Continuously generated news streams transforms search engines in up-to-date newspapers, where breaking news are listed at the top of query results. City-related sensor streams improves our driving experience, with real-time information about navigation, traffic and events.

However, noise can lead to wrong and unexpected results, e.g. errors in traffic sensors lead to wrong representations of the current status of a city, causing—in the worst case—new jams. It is worth noting that noise is inevitable in streaming settings, so technological solutions to cope with it are necessary. A possible way to handle noise is through data integrity constraints, such as data range filters that exclude outliers generated by sensors with hardware failures. However, those methods may fail in detecting noise when data is described according to complex conceptual models. The problem we study in this article is how to assess the *consistency* of streams w.r.t. a fixed and known a-priori conceptual model.

In stream processing scenarios, an important requirement is *responsiveness*: the consistency assessment process must be responsive and quick when analyzing newly arrived data. Batch-based solutions are generally not efficient due to the high latency involved. Furthermore, they have the overhead of processing overlapped data between batches multiple times. Therefore, the stream scenario calls for an incremental solution to cope with the velocity dimension.

The first challenge we cope with is: *how to model the problem of stream consistency checking?* Checking if a (static) dataset is consistent w.r.t. a schema is a well-known problem in data and knowledge management [2, 15]. Moving to the streaming setting, consistency checks in presence of data streams has only been partially considered so far [23]. Such studies assume that the input stream can be managed as a whole. This creates a problem w.r.t. responsiveness, since the whole input stream is needed to produce a result. It follows that methods that compute the output incrementally are desirable in this setting.

In this paper, we consider the DL-lite$_{core}$ description logic to express the schema of data streams. DL-lite$_{core}$ is a member of the well known DL-Lite family, which provides the underpinnings for the standard ontology language OWL2 QL. These logics are expressive enough to cover most features in UML diagrams, and their data complexity for the conjunctive query answering task is AC_0, which is the same as conjunctive querying over relational databases.

The second problem arises from the *volume* of the data: if we consider the set of streams as a whole, the size of the data may be too large to fit in one computing node. We adopt a usual assumption in stream processing settings: the more recent the data, the more relevant it is. This translates in the introduction of windows to capture recent portions of the stream. However, even in this way, the volume of the data may still be too big to be treated in a single machine. A way to overcome this limitation is to perform the consistency check in a distributed fashion. Therefore, a key question is *how to distribute the stream consistency reasoning task while guaranteeing correctness?* This allows to exploit a Distributed Stream Processing Engines (DSPEs), e.g. Apache Flink and Twitter Heron, which require users nto compile the business logics into processing workflows. We focus on how to assess stream consistency on top of DSPEs.

Summarising, the main contributions of this paper include:

1. a formalisation of the stream consistency checking problem w.r.t. a static DL-lite$_{core}$ conceptual schema;
2. two scalable and incremental methods for stream consistency checking. They are designed for a distributed environment. Intermediate reasoning results are cached in the system to check the consistency for newly arrived streams;
3. an analysis of these methods through comparative studies based on the LUBM benchmark [10] and the NPD [21] ontology using Twitter Heron.

2 Preliminaries

In this section we introduce DL-lite$_{core}$ as the Description Logic (DL) we consider in this study. We then describe evolving ontologies, and finally we discuss the distributed stream processing concepts we used to build our solutions.

Static DL-lite$_{core}$ Ontology. DL-lite$_{core}$ is a description logic of the DL-Lite family [3]. Its building blocks are named concepts (denoted with A) and named roles (denoted with P). They can be used to build basic concepts C as $C := \perp \mid A \mid \exists R$; and basic roles R as $R := P \mid P^{-}$. A DL-lite$_{core}$ ontology is composed of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$. The TBox $\mathcal{T}$ contains axioms in the forms of $C_1 \sqsubseteq C_2$ or $C_1 \sqsubseteq \neg C_2$. The former axiom, also known as *Positive Inclusion* (PI), indicates that C_1 is a subclass of C_2, e.g. *Student* is a subclass of *Person* (also denoted $SC(Student, Person)$). The latter one represents a *Negative Inclusion* (NI), which expresses that two classes are disjoint, e.g. there cannot be an instance of *Person* and *Organization* (also denoted $DJ(C_1, C_2)$).

The ABox $\mathcal{A}$ contains assertions about individuals, which can be either $C_k(x_1)$ or $R_k(x_1, x_2)$. If an NI in $\mathcal{T}$ is violated by assertions of $\mathcal{A}$, (e.g. $DJ(C_1, C_2)$ in $\mathcal{T}$, $C_1(a)$ and $C_2(a)$ in $\mathcal{A}$), then the knowledge base is inconsistent. [6] shows that only NI axioms can lead DL-lite$_{core}$ ontologies to inconsistency.

RDF and DL ontology streams. An *RDF stream* $S = ((d_1, t_1), \ldots, (d_n, t_n), \ldots)$ is a potentially unbounded sequence of timestamped informative units (d_i, t_i) ordered by the temporal dimension, where t_i is the timestamp (we consider the time as discrete) and d_i is an RDF statement. An RDF statement is a triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, where I, B, and L identify the sets of IRIs, blank nodes and literals, respectively. An *RDF term* is an element of the set $T = I \cup B \cup L$. RDF streams can be used to serialise DL ontology streams.

We define *DL ontology streams* starting from the notion of *evolving ontology* as defined in [14], which captures the dynamics of a knowledge base. Given a discrete time interval $[m, n]$, a DL ontology stream $\mathcal{O}_m^n$ is a pair $\mathcal{O}_m^n = (\mathcal{T}, \mathcal{A}_m^n)$, where $\mathcal{T}$ is a TBox and $\mathcal{A}_m^n$ is a stream of ABoxes from time m to n. We refer with $\mathcal{A}_m^n(i)$ to the ABox (*snapshot*) at time i associated to $\mathcal{A}_m^n$. Similarly, we refer with $\mathcal{O}_m^n(i)$ to the pair $(\mathcal{T}, \mathcal{A}_m^n(i))$, which is a static ontology.

When handling a stream, it may be important to analyze a portions of data items, e.g., count the occurrences of a fact over a time interval. Inspired by the research on stream and event processing [8], we employ the notion of *window*, which is an operation that selects a portion of items in the stream. Let $\mathcal{O}_m^n$ be a DL ontology stream . The application of a window W over $\mathcal{O}_m^n$ results in:

$$\mathcal{O}_o^c = W(\mathcal{O}_m^n, \omega, c) = (\mathcal{O}_m^n(o), \ldots, \mathcal{O}_m^n(c)) = (\mathcal{T}, \langle \mathcal{A}_m^n(o), \ldots, \mathcal{A}_m^n(c) \rangle) = (\mathcal{T}, \mathcal{A}_o^c),$$

where ω is a natural number representing the *size* of the window, and c is the time on which the window is applied. The following constraints hold: $o = \max\{m, c - \omega\}$ and $c \leq n$. Finally, we define the *window content* as the union of the axioms contained in the stream snapshots, i.e., $u(\mathcal{A}_m^n) = \bigcup_{i=m}^{n} \mathcal{A}_m^n(i)$. Given a DL ontology stream $\mathcal{O}_m^n$, it follows that $\langle \mathcal{T}, u(\mathcal{A}_m^n) \rangle$, is a knowledge base.

Distributed Stream Processing Engines. Processing big amounts of streams usually relies on Distributed Stream Processing Engines (DSPEs). DSPEs provide the advantages of automatic instantiating and distributing tasks onto computing nodes. They also take care of the coordination of the tasks. Users just need to provide the processing logics. In the following, we introduce basic concepts of DSPE, adopting the nomenclature of Apache Storm and Heron.

Fig. 1 (left) shows the processing logics as a *logical topology*, which is a Directed Acyclic Graph (DAG) composed of *spout* and *bolt* nodes. Spouts (S) provide input data to bolts; they emit data to downstream nodes and are typically used to connect a topology with external data sources such as Web services or data brokers. Bolts (B_i) embed the processing logics: they manipulate data from upstream nodes and emit results to downstream nodes.

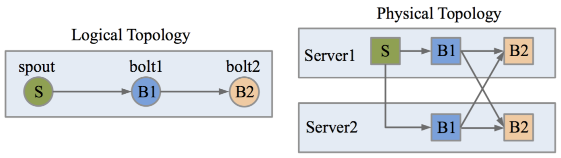


Fig. 1: Deploying a logical topology (left) into a physical (right) one (right). The spout S and bolts $B1$, $B2$ are instantiated as tasks on two computing servers.

Each edge of the topology represents a data stream. Data streams flow through the topology as sequences of *tuples*, i.e. sets of property-value pairs. While defining the logical topology, the user should declare the tuple format (i.e., the set of properties) for every edge.

In addition to the structure of the logical topology, the user should provide configuration information to deploy the logical topology to a computing cluster. First, the user needs to define how many instances of each node should be deployed. These instances are named *task instances* or, shortly, tasks. Moreover, for each edge in the logical topology, the user should define a grouping strategy (e.g., a hashing function), which is used by the DSPE to partition the stream among the tasks. For this reason, given a stream, a subset of the tuple attributes acts as a *key*. The DSPE uses the logical topology and the configuration parameters to decide how to distribute the tasks among the available computing servers. This results in a *physical topology*, as depicted on the right of Fig. 1.

Problem definition. We study how to perform incremental stream consistency checking over DL-lite$_{core}$. That means, given a stream $\mathcal{O}_m^n$ and a time interval of ω time units, we aim to check if $\langle \mathcal{T}, u(W(\mathcal{A}_m^n, \omega, c))\rangle$ is consistent for every $c \in [m, n]$ in an incremental way. It is worth noting that when ω is large and the window starts at the beginning of the input (i.e., $\omega \geq n - m$ and $m = 0$), the problem becomes the incremental consistency assessment over the whole stream.

3 Related Work

We first discuss related studies on consistency checking for knowledge bases. Then, we introduce distributed stream processing and RDF stream processing.

Consistency Checking of an Ontology. Although there have been many studies on consistency checking, e.g., Baclawski et al. [4] translate an ontology to the language that can be executed by a logic programming engine, stream consistency checking is only partially considered. [23] proposes an incremental method that efficiently checks the consistency of new changes against the whole ontology. Our study targets the consistency of streams over DL-lite$_{core}$. Given a static DL ontology, consistency checking is often reduced to query answering. There are various query rewriting techniques that have been proposed [6, 11, 18]. Specially, [6] shows that the computational complexity of a consistency checking task is exponential to the size of TBox at worst. Recently, [16, 17] proposed the adoption of machine learning to achieve fact consistency checking. These methods apply standard reasoning techniques to label inconsistent data instances, and then learn a model to classify new instances. While these approaches are interesting, they cannot guarantee a 100% accuracy. When considering dynamic ontologies, there are some studies on incremental reasoning and evolving ontologies [19, 14]. However, they do not consider distributed stream settings.

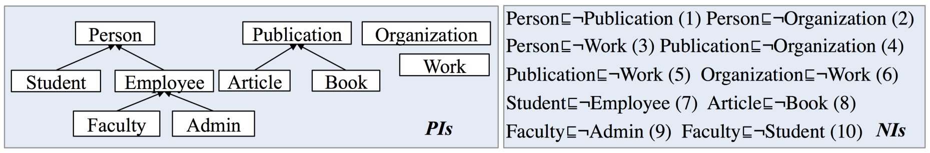
Distributed Stream Processing. Stream processing relies on the idea of managing data *in motion*, by performing tasks in a continuous fashion. This paradigm has recently gained popularity due to the rise of Distributed Stream Processing Engines (DSPEs), which process streams in clusters and cloud services. One of the first DSPEs to gain popularity is Storm [22]. It relies on *topologies*: processing workflows where each element is named *task*. Storm automatically handles the distribution of tasks to computing nodes. Recently, Twitter proposed Heron [12] as a successor of Storm. Heron overcomes some limitations of Storm: the limited performance monitoring, impossibility to deploy in clusters with heterogeneous nodes and complexity in debugging. Other DSPEs exist, each of those with different design goals but with the common idea of workflows, similarly to Apache Storm's topologies. For example, Apache Samza embeds a key-value store to manage state between processing nodes. Apache Flink [7] emphasizes the combination of batched-based and stream-based processing paradigms.

RDF Stream Processing. RDF Stream Processing (RSP) is a recent effort to push the stream processing paradigm in the semantic web. Solutions like C-SPARQL [5] and CQELS [13] adopts *sliding windows* to create time-varying views over the streams, to be processed through SPARQL. Solutions like EP-SPARQL [1] and INSTANS [20] propose to verify time-relation constraints over the stream elements, usually in closed time intervals. The existing RSP solutions have been developed in centralized systems, showing limitations in scalability and in managing data characterized by high velocity and volume.

4 Solution

Given a Tbox, our solutions derive incremental consistency checking procedures to be executed in a DSPE. We first present the running example used in this section, followed by the basic building blocks used in our solutions.

Example 1. Fig. 2 shows the TBox $\mathcal{T}_{ex}$, used as running example and based on the LUBM benchmark. Each node in the figure is a class, while edges denote positive inclusion axioms (PIs), e.g., the axiom $SC(Student, Person)$ is represented by the nodes $Student$, $Person$ and the edge between them. The right part of Fig. 2 contains the negative inclusion axioms (NIs) (e.g., Axiom (1) indicates that an instance of $Student$ cannot be an instance of $Publication$). In total, there are ten classes, six PIs, and ten NIs that are explicitly stated in $\mathcal{T}_{ex}$.



Person⊑¬Publication (1) Person⊑¬Organization (2)
Person⊑¬Work (3) Publication⊑¬Organization (4)
Publication⊑¬Work (5) Organization⊑¬Work (6)
Student⊑¬Employee (7) Article⊑¬Book (8)
Faculty⊑¬Admin (9) Faculty⊑¬Student (10) ***NIs***

Fig. 2: The PIs and NIs of the Tbox $\mathcal{T}_{ex}$ in the running example.

A *tuple* is the basic unit of input, output, and intermediate streams. With reference to the RDF stream model of Section 2, each stream item is a tuple defines as (s, p, o, t). The adoption of DL-lite$_{core}$ implies that a tuple can describe either a role or a class assertion. Class assertions $C(x)$ and $\exists R(x)$ in the snapshot $\mathcal{A}_m^n(t)$ are represented as (x, isA, C, t) and (x, isA, C_R, t), respectively.

A *stream* S_C contains all tuples that state a class assertion of $C(a)$, where a is a generic individual, e.g., S_{Person} is a stream that has instances like $(Bob, isA, Person, t_1)$. $S_{C_{inc}}$ is a special stream that contains the instances found to be inconsistent, e.g., given Axiom (1) in $\mathcal{T}_{ex}$, if there are two instances $(Bob, isA, Person, t_1)$ and $(Bob, isA, Publication, t_1)$, the output tuple $(Bob, isFoundToBe, Inconsistent, t_1)$ is appended to $S_{C_{inc}}$. Role assertions $R(a, b)$ are managed by introducing two class assertions: $\exists R(a)$ and $\exists R^-(b)$, where R^- is the inverse property of R.

Given a TBox $\mathcal{T}$, the *input* to a topology is a set of streams $\mathcal{S} = \{S_{C_1}, \ldots, S_{C_n}\}$, where each stream corresponds to a class in $\mathcal{T}$. The *output* of the topology is $S_{C_{inc}}$ that reports all inconsistent instances. For the sake of illustration, we assume that a topology has only one spout serving as a stream broker, i.e., it collects streams from different sources and emits them as $\mathcal{S}$.

A topology consists of a set of *bolts* $\mathcal{B} = \{B_1, \ldots, B_n\}$. A bolt B_i encodes a set of operations $\{o_1, \ldots, o_n\}$. An operation o_i encodes a part of the consistency check logics: it takes as input a set of streams $\mathcal{S}_{input}^{o_i} = \{S_{C_1}, ..., S_{C_n}\}$ and emits output streams $\mathcal{S}_{output}^{o_i}$. We define two kinds of operations. An *inference operation* $o^{\rightarrow}$ encodes subclass to superclass inferences: it takes one stream S_{C_1} as input and emits one output stream S_{C_2}, i.e., $o^{\rightarrow} : S_{C_1}(x) \rightarrow S_{C_2}(x)$. A *conjunction operation* $o^{\cap}$ is similar to a join function that checks inconsistencies. The input is a set of streams and the output is one stream, i.e., $o^{\cap} : S_{C_1}(x) \wedge \ldots \wedge S_{C_n}(x) \rightarrow S_{C_{output}}(x)$. Differently from inference operations, conjunction ones require the presence of several assertions together to trigger the underlying rule. Therefore, a conjunction operation requires caching instances of each input streams. We exploit windows for this purpose: they cache instances when they arrive and delete them when their associated time instant expires.

For both operations, the *key* should be the value of the instance, e.g., an instance of $S_{Person(Bob)}$ uses "Bob" as a key. This allows the engine to partition the streams and make sure that instances $Person(Bob)$ and $Publication(Bob)$ are processed in the same task instance. The operation output can not serve as input to another operation of the same bolt, i.e., the operations within one bolt are independent of each other, enabling the distribution of workload. Lastly, if an instance does not participate any operations, the bolt simply forwards it to the downstream bolts, which guarantee that all the inconsistencies are detected.

Example 2. Given the $\mathcal{T}_{ex}$ in Example 1, the inference operation $o_i^{\rightarrow}$ derives a new instance of $Person$ for each instance in the $Student$ stream, since $Student$ is a subclass of $Person$, i.e., $o_i^{\rightarrow} := S_{Student}(x) \rightarrow S_{Person}(x)$. Moreover, given the Axiom (1) in Fig. 2, the conjunction operation $o_i^{\cap}$ is defined as $o_i^{\cap} := S_{Person}(x) \wedge S_{Publication}(x) \rightarrow S_{C_{inc}}(x)$. In this case, each stream of S_{Person} and $S_{Publication}$ is associated with a window. When an instance $S_{Person(Bob)}$ arrives, $o_i^{\cap}$ checks whether $S_{Publication(Bob)}$ is cached. If yes, $o_i^{\sqcap}$ outputs $S_{C_{inc}(Bob)}$, since it violates the NI Axiom (1); otherwise, $Person(Bob)$ is cached for future use.

4.1 The NIs Topology Method (NTM)

Fig. 3 depicts three alternative topologies that are compiled from $\mathcal{T}_{ex}$, produced by the two methods we propose. Fig. 3a shows our basic solution, the *NIs Topology Method* (NTM): it computes the set $NI_{closure}$, which contains all the disjointness axioms that can be deduced from the NIs and PIs. Then, it compiles each of them into one bolt as conjunction operations. Specifically, given an NI in the TBox, NTM computes $NI_{closure}$ as the list of subclasses for each class in the NI. The permutation of the subclass elements gives all possible combinations between subclasses, and hence the $NI_{closure}$. Each NI in $NI_{closure}$ is of the form $DJ(C_1, ..., C_n)$, which can be implemented as a conjunction operation. NTM computes topology before runtime since it requires the TBox only.

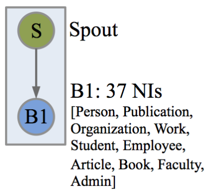
Example 3. The ten NIs in $\mathcal{T}_{ex}$ of Fig. 1 lead to 37 NIs in $NI_{closure}$, e.g., given $DJ(Person, Work)$ and the subclasses of $Person$, NTM infers NIs that are not explicitly stated in $\mathcal{T}_{ex}$: $DJ(Person, Work)$, $DJ(Employee, Work)$, $DJ(Faculty, Work)$, and $DJ(Admin, Work)$. As shown in Fig. 3a, B_1 is the only bolt that performs all the conjunction operations. When deploying this topology, multiple task instances of the bolt partition the stream by the subject field in a tuple, e.g., given a tuple (a, isA, C, t), a is the partition key.

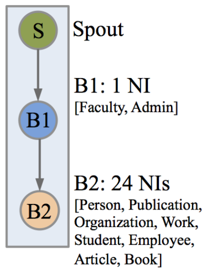
As shown in the experiment, the single bolt of NTM creates a computational bottleneck on the throughput. [6] explains that the size of $NI_{closure}$ is exponential to the size of TBox at worst. Consequently, the number of operations in the bolt may be exponential. We considered two possible remedies to the bottleneck problem, but neither of them solves it completely. A possible remedy is to increase the number of tasks for the bolt. It increases the parallelism, but it cannot reduce the complexity of the bolt. When the stream rate increases, the workload on each task increases as well, which poses the same problem. Another possible remedy is to have a different topology layout. Consider $\mathcal{T}_{ex}$ and a topology where the spout connects to two bolts B_1 and B_2 (not chained in a pipeline): it does not offer any advantages when processing the pairwise PIs between $Person$, $Publication$, $Organization$, and $Work$ (Axioms (1-6)), since these four streams cannot be split into two bolts without duplication.

4.2 The Pipeline Topology Method (LTM)

We propose the *Pipeline Topology Method* (LTM) to improve NTM. LTM arranges the NIs in a hierarchical fashion to reduce their total number, and splits the hierarchy in different bolts to distribute the consistency-checking workload. The bolts in LTM are chained as a pipeline. Each bolt deals with a limited amount of classes and their related NIs. The example below shows our idea.

Example 4. In Fig. 3b, bolt B_1 is assigned to process only two classes $Faculty$ and $Admin$, as well as the only NI between them. In addition, B_1 has two *inference operations* that convert these two streams into one $Employee$ stream


S
Spout
B1
B1: 37 NIs
[Person, Publication, Organization, Work, Student, Employee, Article, Book, Faculty, Admin]
(a) NIs Topology


S
Spout
B1
B1: 1 NI
[Faculty, Admin]
B2
B2: 24 NIs
[Person, Publication, Organization, Work, Student, Employee, Article, Book]
(b) Pipeline Topology 1

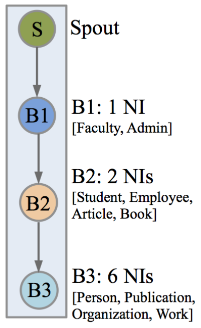

S
Spout
B1
B1: 1 NI
[Faculty, Admin]
B2
B2: 2 NIs
[Student, Employee, Article, Book]
B3
B3: 6 NIs
[Person, Publication, Organization, Work]
(c) Pipeline Topology 2

Fig. 3: The NIs (NTM) and the Pipeline (LTM) topologies. The workload in the last bolt of the three topologies reduces as the length of the pipeline increases.

and output to bolt B_2. B_1 also forwards all other unused streams to B_2. In this way, B_2 does not need to consider *Faculty* and *Admin* anymore. Their related NIs with other classes are still ensured by the disjointness between *Employee* and other classes. Comparing to Fig. 3a, this topology has the cost of adding an extra bolt B_1, however, it greatly reduces the number of NIs in the last bolt from 37 to 24. Assuming each class has the same incoming stream rate, the CPU bottleneck of the last bolt is significantly reduced. Furthermore, the total number of NIs in B_1 and B_2 together is also smaller than the number of NIs in Fig. 3a. The topology in Fig. 3c further extends this idea by using three bolts. Bolt B_2 handles $DJ(Student, Employee)$ and $DJ(Article, Book)$, while bolt B_3 handles the rest of the classes and the six NIs among them. The topology in Fig. 3c has the overhead of forwarding streams through B_1 and B_2.

By comparing the three topologies in Fig. 3, we observe that the excessive amount of NIs in B_1 of Fig. 3a can lead to a computational bottleneck. By arranging the NIs in a hierarchal way, the total number of NIs are significantly reduced. It is possible to use one bolt to hold the entire reduced NI hierarchy. However, it still has the same problem that the single bolt can become a CPU hot-spot. Therefore, by using the operations we defined in Section 2, we split the hierarchy into different bolts, such as in Fig. 3b and 3c. In this way, we achieve that the CPU cost of checking NIs is distributed along the pipeline and alleviate the potential computational bottleneck. Essentially, our LTM method looks for the best way of balancing the workload into a pipeline of bolts.

Algorithm 1 gives the details of LTM. The algorithm takes the TBox $\mathcal{T}$ of the DL ontology stream as an input. The output is a chain (pipeline) of bolts. Each bolt is filled with the operations it needs to perform. The algorithm develops two parts: first, Steps 1-3 find the *essential* NIs and arrange them into hierarchical groups. Steps 4-5 assign these groups to bolts and generate the corresponding operations. Each step of Algorithm 1 is explained in below.

Step 1. As with NTM, Line 1 computes the NI closure set, denoted as $NI_{closure}$.

Step 2. Based on $NI_{closure}$, Line 2 computes NI_{root}, which is a set of all *essential* NIs. The intuition of "essentialness" is that: given two NIs, if each class in one NI is either a sub or an equivalent class of the other, we can first convert the subclass to its superclass, then only process the NI with the superclasses. For example, given Axiom (7) and (10) of Fig. 2, *Faculty* in Axiom (10) is a subclass

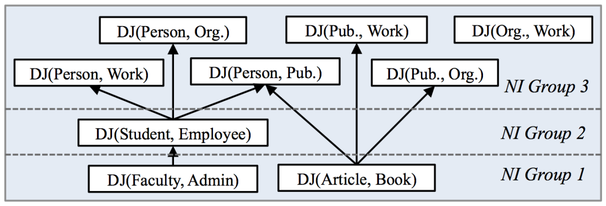
Algorithm 1: CompilePipelineTopology($\mathcal{T}$)

```
input  : $\mathcal{T}$, The TBox of a given DL Ontology
output: $\mathcal{B}$, A pipeline of bolts filled with operations
1  $NI_{closure}$ = ComputeSetOfNIClosure() ;
2  $NI_{root}$=ComputeSetOfNIRoots($NI_{closure}$); $NIGroups$=[];
3  while $NI_{root}$.size() is not reducing do
4      $currNIGroup$ = GetAGroupOfNIs($NI_{root}$);
         $NIGroups$.add($currNIGroup$);
5      $NI_{root}$.removeAll($currNIGroup$);
6  end
7  $NIGroups$.add($NI_{root}$);
8  $NIInBolts$ = AssignNIGroupsToBolts($NIGroups$);
9  $processedClasses$ = []; $\mathcal{B}$= [];
10 for (i = 0; i< $NIInBolts$.size(); i++) do
11     $\mathcal{B}$.add($b_i$ = new Bolt());
12     $b_i$.add(GetConjOps($NIInBolts$[i]), $processedClasses$);
13     $b_i$.add(GetInferOps($NIInBolts$[i], $NIInBolts$[i+1]));
14     $SC$=Set($C_i$ of all $DJ(C_1, ..., C_n) \in NIInBolts[i]$) ;
15     $processedClasses$.add($SC$);
16 end
17 return $\mathcal{B}$
```

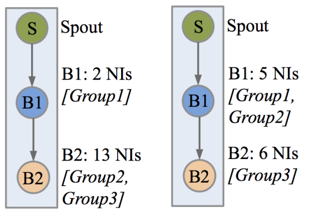
of *Employee* in Axiom (7). Therefore, we can first make an inference operation (i.e., o_i: $Faculty \rightarrow Employee$), and then check Axiom (7) only. By iterating through each pair of NIs in $NI_{closure}$, we can find and keep the essential NIs in NI_{root}. Among the ten NIs in Fig. 2, Axiom (10) is removed from NI_{root}.

Step 3. Lines 3-6 of Algorithm 1 group NIs into a sequential processing order. Intuitively, an NI with no subclasses can be checked first. For example, classes of NI Axiom (9) have no subclasses. It should be checked at the beginning of a pipeline topology (e.g., Place it in bolt B_1 of Fig. 3c. If placed in B_3, its input streams need to be forwarded via B_1 and B_2). These NIs can be found by counting the "in-degree". For example, in Fig. 4a, if there is a sub-to-super relationship between classes of two NIs, we draw an edge between them. The NIs with no incoming edges have zero in-degree. However, not all NIs with zero in-degree should be processed first. For example, $DJ(Organization, Work)$ cannot be processed in bolt B_1 of Fig. 3c, since both *Organization* and *Work* are used again later in the pipeline by other NIs. If placed in B_1, every tuple of $S_{Organization}$ still needs to be forwarded to B_3 for the Axiom (2) $DJ(Organization, Person)$. Therefore, $DJ(Organization, Work)$ is placed together with Axiom (2) at the last group in Fig. 4a. In total, there are three NI groups in Fig. 4a that should be checked in a sequential order.

The algorithm of forming NI groups is adapted from the topological sorting algorithm. It first finds all the NIs in NI_{roots} that fulfill both conditions: 1) has zero in-degree; and 2) its classes are used by the NIs outside the group. These NIs



(a) NI groups



(b) Combining NI groups

Fig. 4: Computing NI groups and assigning them to bolts

are removed from NI_{roots} and form an NI group. Then, the algorithms continue to find another group from NI_{roots} until no group can be found.

Step 4. After Step 3, each NI group can individually form a bolt in the pipeline. This, however, may create too many bolts, and each new bolt incurs extra overhead. To avoid this problem, several adjacent NI groups can be combined into one bolt (Line 8). Fig. 4 gives two ways of combining NI groups to bolts. Fig. 4a contains three NI groups separated by the dashed line. In Fig. 4b, the left topology assigns Group 1 to bolt B_1, and Group 2 and 3 to bolt B_2. The right topology has Group 1 and 2 assigned to B_1, and Group 3 to B_2. The way of combining NI groups impacts system performance, as we discuss at the end of this section.

Step 5. Finally, Line 9-17 compiles the operations ($o^{\rightarrow}$ and $o^{\cap}$) in each bolt.

To compile *conjunction operations* $o^{\cap}$, we compute the closure of NIs in a group to ensure correctness. For example, in the right topology of Fig. 4b, since Group 1 and 2 are combined in bolt B_1, the algorithm first computes their NI closure and then checks each of them (e.g., the closure contains NIs from the two groups together with $DJ(Student, Faculty)$ and $DJ(Student, Admin)$). The process of computing an NI closure has been discussed in NTM. The closure also excludes the classes that have processed in the preceding bolts.

For compiling *inference operations* $o^{\rightarrow}$, we need to find the subclass axioms between the classes of two consecutive bolts (B_i and B_{i+1}). For example, in the left topology in Fig. 4b, bolt B_1 should include the subclass axioms between B_1 and B_2 (e.g., $SC(Faculty, Employee)$ and $SC(Admin, Employee)$). We consider only the *direct* subclass axioms. Axioms like $SC(Faculty, Person)$ are not included, since B_2 handles the NIs between *Employee* and all other classes. Furthermore, bolt B_1 also needs to include all the subclasses of the classes in B_2 that do not participate in the NI roots. For example, if class *Employee* (handled in B_2) has a subclass $FemaleEmployee$, which is not disjoint with either $Faculty$ or $Admin$, it still needs to be inferred at bolt B_1 to become $Employee$ so that the disjointness between $FemaleEmployee$ and other classes are checked in B_2. Each of these subclass axioms can be compiled to an inference operation. Lastly, Algorithm 1 returns a list of bolts that forms a pipeline. Each bolt is filled with the necessary operations to check the consistency.

As mentioned in Step 4, the two ways of combining NI groups have different performance. Based on the actual number of NIs we derived in Step 5 for Fig. 4b, it follows that the right topology distributed the workload more evenly than the left one. The difference of NIs numbers between B_1 and B_2 is much less on the right topology. In the left topology, B_2 will become a bottleneck for the throughput assuming all input streams have the same stream rate. In practice, combining NI groups to bolts is a parameter to be specified by the users. To find the best way of combing NI groups, users can compile different topologies beforehand and choose the one with less skewed workload based on the number of operations. In future, we will extend our work to propose a cost model, where the cost of each operation, the stream rate, the computation, and communication overhead will be considered together to automatically find the best topology.

4.3 Limitations and Discussion

This study considers DL-lite$_{core}$. One of its closest extensions is DL-lite$_{horn}$ [3], which introduces the conjunction of concepts in the left operator, i.e. $C_1 \sqcap ... \sqcap C_n \sqsubseteq C$ The conjunction operation defined above supports this type of axioms. Our above algorithm needs to be extended to cope with it.

LTM has a relatively high memory cost comparing with NTM. The reason is that an instance of a stream can be potentially cached multiple times at different bolts of a LTM topology. For example, in Fig. 3b, stream $S_{Faculty}$ needs to be cached in B_1, since B_1 checks its NI with S_{Admin}. After stream $Faculty$ is inferred as $Employee$ and sent to B_2, its instances need to be cached again in B_2. Our experiments also show this shortcoming. LTM should consider the memory limitation when assigning NI groups to bolts. This work assumes enough memory to emphasize the CPU cost.

5 Experiments

This section first introduces the experiment setup and then report the results.

Setup. As in [23], we use two ontologies for experiment, LUBM [10] and NPD [21]. We adapted their TBoxes to be compliant with DL-lite$_{core}$. LUBM is a well-established benchmark, while NPD is an ontology from the real world. NPD is larger than LUBM: there are 43 streams in the input stream set for LUBM and 329 of them for NPD. In the adapted TBox, LUBM has 56 PIs and 70 NIs; NPD has 332 PIs and 51 NIs. Although NPD has less NIs its closure is bigger than the LUBM's one. We used HermiT [9] to calculate the closure set of PIs.

Both LTM and NTM have one spout, as discussed in Section 4. The generator emits each stream at the same speed, fine tuned to target the system processing capability. We compare the throughput of the topologies by measuring the number of tuples they process in a time interval. We use Heron 0.14.3 as the DSPE; experiments run with 4-6 machines, each of them having 128 GB RAM and two E5-2680 v2 at 2.80GHz processors, with 10 cores per processor.

Overall results. Fig. 5 plots the results for LUBM and NPD. We compare the topology throughput by using two, four, and six computing servers and two bolts for LTM, i.e., LTM topologies have two bolts, B_1 and B_2. A topology LTM-n has the first n NI groups assigned to B_1. Recall that given the three NI groups in Fig. 4a, the LTM-2 topology has the first two NI groups assigned to B_1 and the others to B_2, which results in the right topology in Fig. 4b. Note that LTM-0 places no NI groups in bolt B_1, and has B_2 to check all the NIs. The compiled TBoxes of LUBM and NPD happen to have the same number of five NI groups.

Throughputs are compared under the same amount of tasks for both topologies, e.g. if they are set to have six tasks, in NTM (Fig. 3a) B_1 will have six tasks; in LTM, B_1 and B_2 will have three tasks each (Fig. 3b).

Let's consider the results of LUBM in Fig. 5. First, when using two servers, the throughput of NTM is about twice as much as LTM-0 under different task numbers. Since B_1 in LTM-0 contains no NIs, its three tasks only forward streams to B_2. The three tasks of B_2 in LTM-0 bear the same workload of the six tasks of bolt B_1 in NTM. This explains why the throughput of LTM-0 is roughly half of NTM one. Second, we can observe that LTM-1 greatly improves the throughputs of LTM-0 under different task numbers. This is because LTM-1 distributes the NI-checking workload between B_1 and B_2. Furthermore, the total number of NIs is also reduced, when arranging them in a pipeline. However, the throughputs of NTM and LTM-1 are roughly the same. This suggests that the way LTM-1 distributes the workload cannot outperform NTM. Given the same amount computing resources, B_2 in LTM-1 is still the bottleneck. Third, LTM-2 shows the best performance: it outperforms NTM by up to 139% with six tasks. This suggests that LTM-2 topology distributes NIs more evenly (there are two NI groups assigned to B_1 of LTM-2) than LTM-0 and LTM-1, and LTM can perform better than NTM. Fourth, moving from LTM-2 to LTM-5 the throughput decreases, since more NI groups are placed onto B_1 than on B_2 and lead to an unbalanced situation. LTM-5 has all NI groups assigned to B_1, therefore, its performance is similar to that of LTM-0. The change of throughput from LTM-0 to LTM-5 shows that evenly distributing the NI-checking workload can avoid the bottleneck bolt slowing down the overall performance.

Looking at NPD in Fig. 5, the trend of throughput is similar to LUBM: it grows from LTM-0 to LTM-3 and then decreases. NPD always has the best performance at LTM-3 topology: it follows that LTM-3 has the most evenly distributed workload. The throughput improvement of LTM over NTM in the NPD case is up to 330%, when LTM-3 topology has ten tasks running on six servers. The improvement ratio is higher than that of the LUBM case: the reason is that NPD has a much larger TBox. When arranging its NIs into a hierarchy, LTM saves more conjunction operations by replacing them with inference operations.

Operations Breakdown and Memory Cost. Fig. 6a and 6b plot two metrics to further investigate the reasons of the above results. Since the results are similar on LUBM and NPD, we focus on the former for brevity. Fig. 6a gives the breakdown of the number of conjunction operations in each bolt. NTM, LTM-0, and LTM-5 have the same amount of 773 conjunction operations (NIs) executed

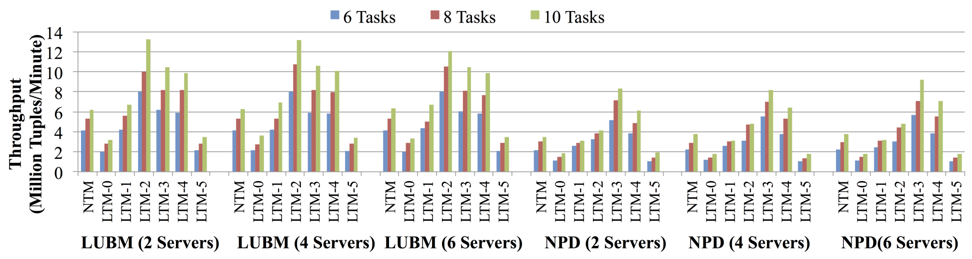
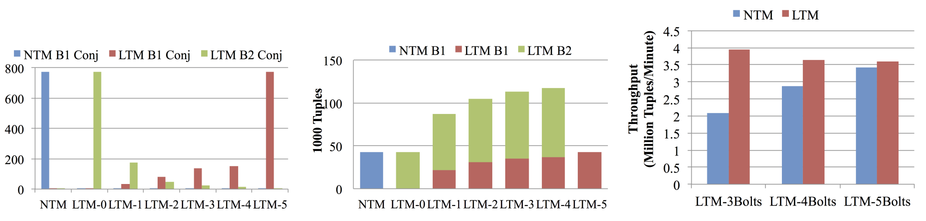


Fig. 5: Results of a NTM topology and LTM topologies with two bolts. The y axis gives the throughput (number of tuples per minutes); the x axis denotes the topologies: NTM stands for the NTM topology, LTM-n denotes a LTM topology.



(a) Operations breakdown (b) Memory cost (c) LTM with n bolts

Fig. 6: Conjunction operations breakdown, memory cost, and LTM with n bolts.

in one bolt, since the closure of NIs is computed by using all the classes. The plot shows that the conjunction number of B_1 (green bar) decreases as the number of B_2 increases (red bar). LTM-2 has the smallest difference between B_1 and B_2, suggesting the reasons for its best throughput. The number of inference operations is relatively small (around 20), so it is not plotted in the figure.

Fig. 6b reports the memory cost (as the number of tuples cached). NTM, LTM-0 and LTM-5 have the same memory footprint, since they have similar NTM workloads. LTM-(1-4) incur higher memory costs than NTM, LTM-0 and LTM-5. As discussed in Section 4.3, it reflects one shortcoming of LTM that some instances need to be cached multiple times in the pipeline, e.g., in Fig. 3b, an instance of $Faculty$ is cached at B_1, and after B_1 infers $Employee$, the instance needs to cached again in B_2. Furthermore, the memory footprint grows from LTM-1 to LTM-5. As B_1 handles more classes and inference operations, more instances need to be cached in B_2. Hence, the total memory cost grows.

Results of LTM with multiple bolts. Fig. 6c compares NTM and LTM topologies with different number of bolts (using three servers). We increase the bolts number for LTM from three to five (each bolt has one task). The task number of B_1 in NTM is set to be three, four, and five. We fine-tuned the NI groups for LTM to give the best performance. Fig. 6c shows that the throughput of LTM decreases when using more bolts. This is because the communication cost increases. Moreover, the throughput of NTM increases linearly with the number of tasks. This suggests that the benefits of reducing NIs might be offset by the communication cost when the length of a pipeline topology grows.

6 Conclusions

While consistency checking is usually studied in the context of static knowledge bases, in this paper, we focus on the problem of incremental consistency checking over streams and propose scalable solutions that can be deployed on a DSPE. More precisely, our two methods can compile a DL-lite$_{core}$ TBox into a processing workflow of a DSPE. The baseline method NTM adapts techniques such as NIs query rewriting to generate continuous queries to assess the consistency. However, NTM involves an excessive number of operations that slow down its performance. To overcome this issue, we propose LTM, where the workload of consistency checking is distributed across a pipeline. This leads to a reduction of the CPU overhead and an improvement of the throughput. Our experiment results show that LTM outperforms NTM by up to 139% and 330%, based on the LUBM benchmark and the NPD ontology, respectively.

References

1. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: WWW. pp. 635–644. ACM (2011)
2. Arenas, M., Bertossi, L.E., Chomicki, J.: Consistent Query Answers in Inconsistent Databases. In: PODS. pp. 68–79. ACM Press (1999)
3. Artale, A., Calvanese, D., Kontchakov, R., Zakharyaschev, M.: The dl-lite family and relations. CoRR abs/1401.3487 (2014)
4. Baclawski, K., Kokar, M.M., Waldinger, R.J., Kogut, P.A.: Consistency checking of semantic web ontologies. In: ISWC '02. pp. 454–459. ISWC '02 (2002)
5. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Querying RDF streams with C-SPARQL. SIGMOD Record 39(1), 20–26 (2010)
6. Botoeva, E., Artale, A., Calvanese, D.: Query Rewriting in DL-Lite$_{horn}^{(HN)}$. In: Description Logics. CEUR Workshop Proceedings (2010)
7. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink$^{\mathrm{TM}}$: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull. 38(4), 28–38 (2015)
8. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. 44(3), 15 (2012)
9. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: HermiT: An OWL 2 Reasoner. J. Autom. Reasoning 53(3), 245–269 (2014)
10. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Sem. 3(2-3), 158–182 (2005)
11. Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyaschev, M.: The combined approach to query answering in dl-lite. pp. 247–257. KR'10 (2010)
12. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K., Taneja, S.: Twitter Heron: Stream Processing at Scale. In: SIGMOD. pp. 239–250 (2015)
13. Le Phuoc, D., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: ISWC'11. pp. 370–388 (2011)
14. Lecue, F., Pan., J.Z.: Consistent Knowledge Discovery from Evolving Ontologies. In: Proc. of AAAI'15. (2015)
15. Lembo, D., Ruzzi, M.: Consistent Query Answering over Description Logic Ontologies. In: Description Logics (2007)
16. Paulheim, H., Gangemi, A.: Serving dbpedia with dolce — more than just adding a cherry on top. In: ISWC. pp. 180–196 (2015)
17. Paulheim, H., Stuckenschmidt, H.: Fast approximate a-box consistency checking using machine learning. In: ISWC. pp. 135–150 (2016)
18. Pérez-Urbina, H., Motik, B., Horrocks, I.: A comparison of query rewriting techniques for dl-lite. In: Proceedings of DL'09 (2009)
19. Ren, Y., Pan, J.Z., Guclu, I., Kollingbaum, M.J.: A combined approach to incremental reasoning for EL ontologies. In: RR'16. pp. 167–183 (2016)
20. Rinne, M., Solanki, M., Nuutila, E.: Rfid-based logistics monitoring with semantics-driven event processing. In: DEBS. pp. 238–245 (2016)
21. Skjæveland, M.G., Lian, E.H., Horrocks, I.: Publishing the norwegian petroleum directorate's factpages as semantic web data. In: ISWC. pp. 162–177 (2013)
22. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D.V.: Storm@twitter. In: SIGMOD. pp. 147–156 (2014)
23. Wu, J., Lécué, F.: Towards Consistency Checking over Evolving Ontologies. In: CIKM. pp. 909–918. ACM (2014)