COSE: Continuous Shapes Extraction from Dynamic Knowledge Graphs

Baya Dhouib
EURECOM
Biot, France
baya.dhouib@eurecom.fr

Raja Appuswamy
EURECOM
Biot, France
raja.appuswamy@eurecom.fr

***Abstract*—Given the rise in the popularity of Knowledge Graphs (KGs) in various domains, there has been a growing demand for tools that can validate KGs and ensure data quality. Shapes constraint languages, such as SHACL and ShEx, have emerged to provide this functionality, and techniques have been developed to automatically extract validating shapes from KGs. However, to our knowledge, all existing shapes extraction techniques primarily target static KGs and are incapable of dealing with a more dynamic scenario where KGs evolve due to continuous updates. In such a context, it is necessary not just to use generated constraints to validate new updates (data repair) but also to continuously generate constraints themselves (constraint repair).**

In this work, we introduce the COntinuous Shapes Extraction (COSE) framework, which facilitates the continuous validation of KGs by dynamically generating and updating shape constraints in response to changes. COSE identifies updates between KGs versions, generates a delta graph based on these updates, extracts delta shapes from the graph, and integrates them with existing shapes to maintain data integrity. Using several versions of three real-world KGs, we perform a comparative evaluation to demonstrate that COSE can provide up to 10×–28× reduction in shapes extraction time compared to other state-of-the-art solutions while maintaining high accuracy. Finally, we identify several open problems that merit further attention related to the task of continuous KGs validation.

***Index Terms*—Knowledge Graphs, SHACL, Shapes Extraction, Incremental Validation, Evolving Graphs**

I. Introduction

Over the past few years, Knowledge Graphs (KGs) have witnessed widespread adoption in various domains, including enterprise data management, web services, and artificial intelligence. As KGs evolve, they undergo frequent modifications, additions, deletions, and updates of triples, which can lead to inconsistencies, redundancies, and errors that compromise the integrity of the data stored in these graphs. Consequently, there is a growing demand for tools and techniques that can perform quality assessment and validation in a dynamic, ever-changing environment.

Shapes constraint languages such as SHACL [1] and ShEx [2] have emerged as formal frameworks for specifying and enforcing data integrity rules over RDF graphs, much like relational integrity constraints for databases. For instance, SHACL allows users to express validation rules in a separate "shapes graph" and then generate a detailed validation report of every violation. These shapes enable a variety of validations, including enforcing that certain properties must be present (cardinality constraints), that property values conform to specific datatypes or classes (datatype and class constraints), and that relational structures (paths) adhere to domain-specific rules (path constraints). Traditionally, these shapes have been handcrafted by domain experts—a process that becomes untenable as KGs grow in size and complexity.

In response, several systems such as SheXer [3] and QSE [4] automatically infer validating shapes from a single static snapshot of a KG. All existing shape-extraction tools focus exclusively on static graphs [5], [6]. However, many large KGs such as Wikidata, DBpedia, and numerous domain-specific knowledge repositories undergo real-time or near-real-time updates. In these settings, new triples may be inserted or modified hourly or daily rather than in infrequent, large "batch" releases. For example, during the COVID-19 pandemic, Wikidata served as a central repository for global, real-time updates on case counts, vaccine statistics, and government policies. Dashboards driven by Wikidata frequently suffered from conflicting case numbers or missing vaccine approval dates, demonstrating that a one-off shapes extraction cannot maintain data consistency as new information arrives [7], [8]. Similarly, OpenStreetMap (OSM) updates continuously during disaster response—for instance, reporting new road closures or shelter locations. In one documented incident, erroneous OSM edits delayed flood-relief because no incremental validation was in place to catch a missing "roadClosed" tag [9]. These examples underline that continuous updates demand an incremental validation strategy.

Unfortunately, current shapes extraction tools are not designed to deal with incremental updates. Therefore, the only solution today is to execute shapes extraction on a full graph each time an update is released. Such an approach has two problems. First, since the runtime of shapes extractors is proportional to the graph size, this approach of full-validation-at-each-release does not scale well as the graph increases in size across generations. Second, even if only a tiny fraction of triples change between two versions of a graph, shapes extraction requires reprocessing the entire graph, resulting in wasted computational resources.

A more practical approach to dealing with incremental updates is to perform *continuous shapes extraction*. In such a scenario, we have a validated set of shapes $\mathcal{S}_n$ for version

V_n. When the graph evolves to V_{n+1}, the fundamental question becomes: can we generate just the "delta shapes" from the new or modified triples $(V_{n+1} \setminus V_n)$ and then merge them with $\mathcal{S}_n$, rather than re-extracting from scratch? To our knowledge, there has been no work in the literature that has investigated the applicability of state-of-the-art shapes extraction tools in this continuous shapes extraction context, or evaluated the effectiveness of continuous shapes extraction in handling updates to public KGs.

In this paper, we make the following contributions:

- We present COSE, an end-to-end framework for continuous shapes extraction over evolving knowledge graphs. Unlike existing tools such as QSE [4], which are designed for static graphs, COSE operates incrementally, processing only the updates between graph versions. Given an initial version V_n with precomputed shapes, COSE extracts shapes for the next version V_{n+1} in three steps: (1) it detects structural changes to construct a *delta graph*, (2) it expands this delta graph via *type-specific sampling* to recover contextual information necessary for shape inference, and (3) it applies QSE to extract *delta shapes* from the enriched delta graph. These delta shapes are then merged with previously extracted shapes, using a conflict resolution strategy that merges compatible shapes automatically and flags unresolved conflicts for user inspection. By working over deltas and incorporating sampling, COSE achieves substantial speedups while maintaining high accuracy, making it well-suited for continuous or high-frequency KG updates where full recomputation is infeasible.
- Using several versions of three real-world KGs (DBpedia, YAGO, Wikidata), we perform a comparative evaluation to understand the performance–accuracy trade-offs in continuous shapes extraction with COSE. Our analysis demonstrates that COSE can scale well to large graph sizes and provide up to 10×–28× reduction in execution time compared to static shapes extraction while maintaining high accuracy.

II. Design

In this section, we present the COSE pipeline for incremental shapes extraction that consists of six phases as shown in Figure 1. Algorithm 2 summarizes the end-to-end incremental procedure, and table I lists the notation used throughout this section.

A. Baseline Setup

COSE's pipeline begins with a full snapshot V_n of the KG. COSE invokes QSE [4] on V_n to generate SHACL shapes. We provide an overview of QSE's algorithm and data structures that are relevant to our discussion here and refer the reader to prior publication about QSE [4] for further information. QSE scans each triple $(s,p,o) \in V_n$ once, inferring $c = \text{classOf}(s)$ to increment $\Psi^n_{\text{cec}}[c]$, computing $\mathit{objType} = \text{typeOf}(o)$ to add $\mathit{objType}$ to $\Psi^n_{\text{cpot}}[c][p]$, and, if s has not yet been counted for the triplet $(c,p,\mathit{objType})$, incrementing $\Psi^n_{\text{sts}}[(c,p,\mathit{objType})]$.

TABLE I: Notation

Symbol	Description
V_n	Set of RDF triples in version n.
V_{n+1}	Set of RDF triples in version $n+1$.
N_n	$\lvert V_n \rvert$, number of triples in V_n.
N_{n+1}	$\lvert V_{n+1} \rvert$, number of triples in V_{n+1}.
changes	Set of inserted/updated triples.
M	$\lvert \mathit{changes} \rvert$, number of changed triples.
F	$\lvert \mathit{filtered} \rvert$, number of triples passing validation.
V_δ	Enriched delta graph (filtered + sampled context).
D	$\lvert V_\delta \rvert$, number of triples in the delta graph.
S_n	SHACL shapes extracted on version n.
S_δ	Set of "delta shapes" extracted from V_δ.
$\lvert\Psi\rvert$	Total number of entries across auxiliary maps Ψ_{cec}, Ψ_{cpot}, and Ψ_{sts}.

Here, Ψ^n_{cec} records, for each class c, how many entities appear with type c; Ψ^n_{cpot} tracks, for each (c,p) pair, all object types seen; and Ψ^n_{sts} counts support for each $(c,p,\mathit{objType})$ pattern. Once built, QSE extracts one SHACL `NodeShape` per key $(c,p,\mathit{objType})$, computing $\mathit{support} = \Psi^n_{\text{sts}}[(c,p,\mathit{objType})]$ and $\mathit{confidence} = \mathit{support}/\Psi^n_{\text{cec}}[c]$.

Finally, we modified QSE so that it serializes Ψ^n_{cec}, Ψ^n_{cpot}, and Ψ^n_{sts} to disk via Kryo [10], which provides compact binary serialization and fast read–write performance, allowing the Ψ maps to be persisted efficiently across versions. Step 1 of continuous shapes extraction shown in Figure 1 assumes these precomputed maps Ψ^n_{cec}, Ψ^n_{cpot}, and Ψ^n_{sts} built by QSE on V_n are available.

B. Change Detection

COSE treats each new release V_{n+1} as a delta over the prior version V_n. Some KGs publish only the incremental updates; others publish full dumps. If an incremental dump is available, COSE loads it directly. Otherwise, it identifies changes by one of three methods (Algorithm 1).

In the first scenario, both V_n and V_{n+1} are available as full graphs. In this case, COSE performs an external merge-diff. Both graphs are sorted on the composite key $(\mathit{subject},\mathit{predicate},\mathit{object})$. Two iterators, `Old` and `New`, then scan the sorted streams in tandem. Whenever `Old.current = New.current`, both advance. If a triple (s,p,o_{new}) appears in V_{n+1} but not in V_n (or is lexicographically smaller), it is appended to *changes*. If (s,p) matches but $o_{\text{new}} \neq o_{\text{old}}$, COSE treats it as a modification and adds (s,p,o_{new}), while the "old" triple (s,p,o_{old}) is recorded in a separate deletions list. Triples present in V_n but absent from V_{n+1} (pure deletions) are similarly collected in the deletions list but are otherwise skipped.

In the second scenario, if explicit edit logs are available (e.g., in Wikidata), COSE parses the log in a single pass. The triple from each INSERT event is added to *changes*. Each UPDATE $(s,p,o_{\text{old}} \to o_{\text{new}})$ also contributes (s,p,o_{new}) to *changes* and (s,p,o_{old}) to the deletions list. DELETE events are recorded in the deletions list but are not added to *changes*.

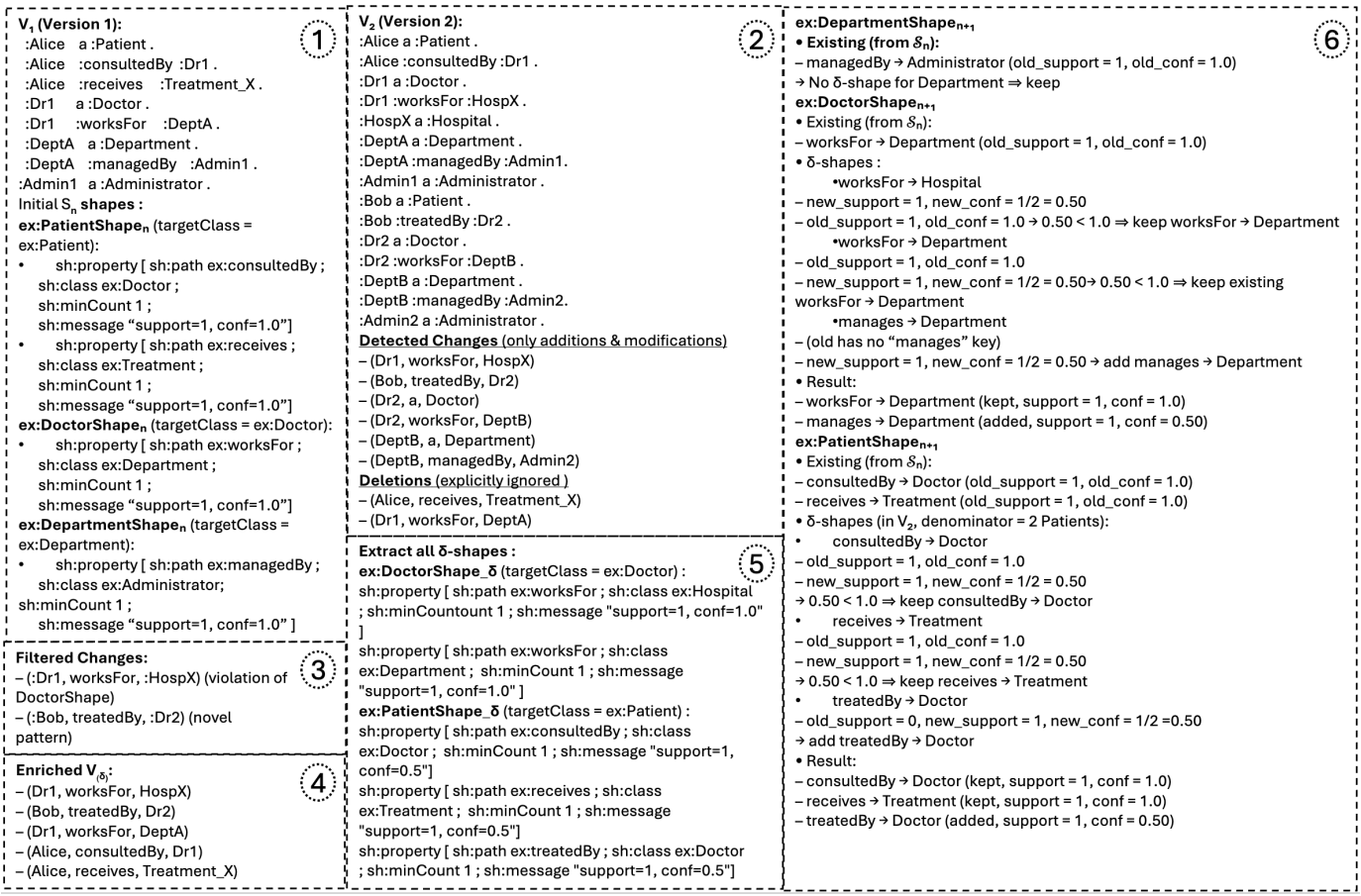


Fig. 1: Overview of the six phases of COSE: ① Data Preparation, ② Change Detection, ③ Filtering by Validation, ④ Delta Graph Generation, ⑤ Delta Shapes Extraction, ⑥ Shapes Merging and Conflict Resolution.

In the third case, when only a SPARQL endpoint is accessible, COSE issues two queries. The first, `SPARQLminus`(V_n, V_{n+1}, ADD), returns all $(s,p,o) \in V_{n+1} \setminus V_n$. The second, `SPARQLvalueDiff`$(V_n, V_{n+1})$, retrieves each $(s,p,o_{\text{old}} \to o_{\text{new}})$ where the object changed. From each returned update, (s,p,o_{new}) is appended to *changes* and (s,p,o_{old}) is added to the deletions list.

In all cases, by the end of this step, *changes* contains precisely those triples that must be added or updated ($|changes| = M$), and the deletions list records any removed triples. Pure deletions do not introduce new class–predicate patterns and therefore cannot yield new shapes, so they are omitted from Δ. They are nevertheless retained to keep the Ψ-maps numerically consistent when support and class-level statistics are updated during serialization. Step 2 of Figure 1 shows V_{n+1} adds the triples $(:Bob, rdf:type, :Patient)$, $(:Bob, treatedBy, :Dr_2)$, $(:Dr_2, rdf:type, :Doctor)$, $(:Dr_2, worksFor, :Dept_B)$, $(:Dept_B, rdf:type, :Department)$, $(:Dept_B, managedBy, :Admin_2)$, modifies $(:Dr_1, worksFor, :Dept_A) \to (:Dr_1, worksFor, :Hosp_X)$, and deletes (ignored in Δ) $(:Alice, receives, :Treatment_X)$, $(:Dr_1, worksFor, :Dept_A)$

C. Filtering Optimization

Although *changes* records all updates to the KG, many updates might simply cause no changes to the shape constraints $\mathcal{S}_n$. For example, if $\mathcal{S}_n$ already enforces that every Person must have a `hasName` property of type `xsd:string`, then adding the triple `(:Alice, hasName, "Alice"8sd:string)` introduces neither a new $(\text{Person}, \texttt{hasName})$ pattern nor a violation, so it can be discarded. Thus, an optimization COSE applies is to identify such updates and eliminate them from further processing. Concretely, for each $u = (s,p,o) \in changes$, we compute $(c,p) = (\text{classOf}(s), p)$ (using s's `rdf:type`) and check via a hash lookup whether (c,p) already appears among the $(\text{targetClass}, \text{path})$ keys in $\mathcal{S}_n$. If it does not, u introduces a new class–predicate pattern and is kept (we also insert (c,p) into the hash set so future triples with that same pattern are dropped). If it does appear, then we run Jena's SHACL validator to validate u against $\mathcal{S}_n$. If u violates a SHACL constraint, we keep u. If there is no violation, we discard u.

COSE employs a filtering heuristic to discard updates unlikely to affect shape extraction, such as those not introducing new class–predicate patterns or modifying schema-relevant properties. To avoid cumulative errors, it conservatively filters only updates involving predicates already covered by existing

Algorithm 1: DetectChanges

Input: V_n, V_{n+1}
Output: *changes*, *deletions*

changes $\leftarrow \emptyset$; *deletions* $\leftarrow \emptyset$;
if *isEditLog*(V_{n+1}) **then**
  foreach *event e in parseEditLog*(V_{n+1}) **do**
    if *e is INSERT* **then**
      changes.add((s,p,o_{new}));
    else if *e is UPDATE* $(s,p,o_{old} \rightarrow o_{new})$ **then**
      changes.add((s,p,o_{new})); *deletions*.add((s,p,o_{old}));
    else
      deletions.add((s,p,o));
  return *changes*, *deletions*
else if *isSPARQLEndpoint*(V_{n+1}) **then**
  foreach (s,p,o) *in SPARQLminus*(V_n, V_{n+1}, *ADD*) **do**
    changes.add((s,p,o));
  foreach $(s,p,o_{old} \rightarrow o_{new})$ *in SPARQLvalueDiff*(V_n, V_{n+1}) **do**
    changes.add((s,p,o_{new})); *deletions*.add((s,p,o_{old}));
  return *changes*, *deletions*
else
  SortDumpOnKey(V_n); SortDumpOnKey(V_{n+1}); Old $\leftarrow$ openSortedStream(V_n); New $\leftarrow$ openSortedStream(V_{n+1});
  while *not end of both streams* **do**
    $t_{\text{old}} \leftarrow$ Old.current() (if exists); $t_{\text{new}} \leftarrow$ New.current() (if exists); **if** $t_{old} = t_{new}$ **then**
      Old.advance(); New.advance();
    else if t_{new} *present and* t_{old} *absent* ***or*** $(s,p,o_{new}) < (s,p,o_{old})$ **then**
      changes.add(t_{new}); New.advance();
    else if $(s_{old}, p_{old}) = (s_{new}, p_{new})$ *and* $o_{old} \neq o_{new}$ **then**
      changes.add($(s_{\text{new}}, p_{\text{new}}, o_{\text{new}})$); *deletions*.add($(s_{\text{old}}, p_{\text{old}}, o_{\text{old}})$); Old.advance(); New.advance();
    else
      deletions.add(t_{old}); Old.advance();
  return *changes*, *deletions*

shapes and omits changes to literals without type annotations. Empirical results (section III) suggest that this filtering introduces minimal deviation in extracted shape quality.

D. Delta Graph Generation

Filtered changes are then used to build a delta graph V_δ. We first add any filtered triple u to V_δ. While these triples supply additions and modifications, they alone are insufficient for shapes extraction, as support and confidence metrics computed by QSE to identify non-spurious shapes depend on the statistical co-occurrence of types, properties, and object types. Therefore, V_δ must not only represent the filtered changes but must also be enriched to include entities of the same class and their associated properties to provide the necessary context. COSE provides three possible strategies to perform this enrichment.

The Type-Specific Sampling (TSS) strategy proceeds as follows. For each class c among the subjects of V_δ, we retrieve
$E_c = |\Psi^n_{\text{cec}}[c].\text{entities}|$, then compute $N_c = \max\big(\lceil \alpha \times E_c \rceil, k_{\min}\big)$, draw N_c entities uniformly at random without replacement from that pool, and for each sampled entity e append all its $\Delta(c)$ property–object-type pairs (from $\Psi^n_{\text{cpot}}[c]$) to V_δ. Here, F is the number of filtered triples already in V_δ, and $\Delta(c)$ is the number of triples each sampled entity of class c contributes. Summing over all classes yields $D = F + \sum_c N_c \Delta(c)$. Step 4 of Figure 1 enriches V^δ_n by re-inserting $(:Dr_1, worksFor, :Dept_A)$, $(:Alice, consultedBy, :Dr_1)$, $(:Alice, receives, :Treatment_X)$, so $V^\delta_n = \{(:Dr_1, worksFor, :Hosp_X), (:Bob, treatedBy, :Dr_2), (:Dr_1, works{-}For, :Dept_A), (:Alice, consultedBy, :Dr_1), (:Alice, receives, :Treatment_X)\}$.

The second enrichment strategy is Full Sampling (FS). For each class c we simply set $N_c = E_c$, meaning every c-typed entity is added to V_δ along with all its property–object-type pairs.

Finally, in our third enrichment strategy, which we refer to as No Enrichment (NS), we set $N_c = 0$ for every class, so no additional context beyond the F filtered triples is included. Consequently, V_δ contains only those F triples.

The use of type-aware sampling introduces a probabilistic bias toward more frequently updated types. While this helps retain representative patterns in practice, it may under-sample rare shape structures. Assuming a uniform random sampling per type with sampling ratio ε, the expected support for a constraint in the sampled delta graph is $\varepsilon \cdot \text{support}_{\text{full}}$. Thus, setting thresholds too high may suppress low-frequency but valid shapes. In our design, we mitigate this by using relaxed support thresholds during delta shape extraction and by merging overlapping patterns incrementally.

E. Delta Shapes Extraction

With V_δ enriched, COSE invokes QSE on V_δ to obtain the candidate delta-shape set $\mathcal{S}_\delta$. Step 5 of Figure 1 runs QSE on V^δ_n to extract $\mathcal{S}_\delta$.

F. Shapes Merging and Conflict Resolution

COSE now merges the candidate delta shapes $\mathcal{S}_\delta$ into the existing full-graph shape set $\mathcal{S}_n$ to produce $\mathcal{S}_{n+1}$. First, we make a $\mathcal{S}_{n+1}$ as a shallow copy of $\mathcal{S}_n$. Next, for each shape $s \in \mathcal{S}_\delta$, we compute its unique key $k = (c, p)$ based on s's target class c and path p. If k does not yet exist in $\mathcal{S}_{n+1}$, then s is truly new and can be inserted. If k already exists, we compare s against the existing shape $s_{\text{old}} = \mathcal{S}_{n+1}[k]$. When s_{old} and s agree on object-type, support, and confidence, s is considered overlapping and can be safely ignored.

If, however, s differs from s_{old} in object-type or in its (support, confidence) pair, we must decide which version to keep. To do so, we compute a "significance" score for both s and s_{old}. First, let $support(s)$ be the number of $(c, p, \text{objType})$ occurrences in V_δ and $confidence(s) = support(s)/\Psi^\delta_{\text{cec}}[c]$. We then normalize support by the maximum support observed over all shapes in $\mathcal{S}_n \cup \mathcal{S}_\delta$:

$$support_{\max} = \max_{s' \in \mathcal{S}_n \cup \mathcal{S}_\delta} support(s')$$

$$sup\_norm(s) = \frac{support(s)}{support_{\max}}, \quad conf\_norm(s) = confidence(s).$$

Because confidence is already a fraction in [0, 1], there is no need to normalize it. Given user-specified weights $\alpha_{\text{sup}}, \alpha_{\text{conf}} > 0$ (default 1:1), we define

$$\text{significance}(s) = \frac{\alpha_{\text{sup}}\, sup\_norm(s) + \alpha_{\text{conf}}\, conf\_norm(s)}{\alpha_{\text{sup}} + \alpha_{\text{conf}}}.$$

We compute the same for s_{old}. If $\text{significance}(s) > \text{significance}(s_{\text{old}})$, then s_{old} is replaced by s; otherwise, s is marked unresolved. Unresolved shapes do not block downstream processing. Step 6 of Figure 1 merges $\mathcal{S}_\delta$ into $\mathcal{S}_n$: Department's $(:Department, managedBy, :Administrator)$ stays; Doctor's $(:Dr_1, worksFor, :Dept_A)$ stays and $(:Dr_1, worksFor, :Hosp_X)$ is added; Patient's $(:Alice, consultedBy, :Dr_1)$ and $(:Alice, receives, :Treatment_X)$ stay and $(:Bob, treatedBy, :Dr_2)$ is added.

While the merging strategy described above uses statistical significance to resolve conflicts, it does not account for semantic consistency. In particular, incompatible object types (e.g., a property inferred as both `xsd:integer` and `xsd:string`) may result in contradictory constraints that always trigger validation errors. To help identify such cases, COSE provides an optional post-merge validation step, which evaluates merged shapes over sampled instances and records any inconsistencies. Conflicting patterns that cannot be reconciled are flagged for manual inspection. Incorporating domain-specific knowledge (e.g., ontologies or type hierarchies) to guide future merging decisions remains an area for future exploration.

G. Serialization

After merging, COSE updates the three Ψ-maps used by QSE to reflect the graph changes and persist them for the next version. `incUpdateMaps` receives Ψ^n_{cec}, Ψ^n_{cpot}, Ψ^n_{sts}, the unified change list *changes*, and the deletions list. It processes each inserted or modified triple $(s,p,o) \in changes$ by incrementing class counts, property–object-type counts, and support counters. Similarly, each deletion (s,p,o_{old}) not paired with an update decrements the corresponding Ψ_{sts}, Ψ_{cec}, and Ψ_{cpot} entries. Once the new Ψ-maps and merged shape set $\mathcal{S}n+1$ are ready, COSE serializes the maps via Kryo. We have modified QSE so that it can directly use these maps as the baseline state for loading in $Vn+2$ without having to recompute them from the full graph.

Algorithm 2: COSE Incremental Shapes Extraction

Input: $V_n, \mathcal{S}_n, V_{n+1}, \alpha, k_{\min}$
Output: $\Psi^{n+1}_{\text{cec}}, \Psi^{n+1}_{\text{cpot}}, \Psi^{n+1}_{\text{sts}}$, new $\mathcal{S}_{n+1}$

if *no serialized Ψ-maps exist* **then**
  $(\Psi^n_{\text{cec}}, \Psi^n_{\text{cpot}}, \Psi^n_{\text{sts}}) \leftarrow \text{QSE}(V_n)$
else
  $(\Psi^n_{\text{cec}}, \Psi^n_{\text{cpot}}, \Psi^n_{\text{sts}}) \leftarrow \text{loadMaps}()$
$(\text{changes}, \text{deletions}) \leftarrow \text{DetectChanges}(V_n, V_{n+1})$; $V_\delta \leftarrow \emptyset$;
foreach *triple u in changes* **do**
  if *violatesSHACL*$(u, \mathcal{S}_n)$ ***or*** *novelPattern*$(u, \mathcal{S}_n)$ **then**
    V_δ.add(u)
foreach *class c in subjects*(V_δ) **do**
  pool $\leftarrow \Psi^n_{\text{cec}}[c]$.entities; $N_c \leftarrow \max(\lceil \alpha \cdot |\text{pool}| \rceil, k_{\min})$;
  sampleSet $\leftarrow$ randomSample(pool, N_c);
  foreach *entity e in sampleSet* **do**
    foreach $(prop, objType)$ *in* $\Psi^n_{\text{cpot}}[c]$ **do**
      V_δ.add$(e, \text{prop}, \text{objType})$
$(\Psi^\delta_{\text{cec}}, \Psi^\delta_{\text{cpot}}, \Psi^\delta_{\text{sts}}) \leftarrow \text{QSE}(V_\delta)$; $\mathcal{S}_\delta \leftarrow \emptyset$;
foreach $(c, prop, objType)$ *in keys*$(\Psi^\delta_{\text{sts}})$ **do**
  support $\leftarrow \Psi^\delta_{\text{sts}}[(c, \text{prop}, \text{objType})]$;
  confidence $\leftarrow$ support $/\, \Psi^\delta_{\text{cec}}[c]$;
  $s \leftarrow$ buildShape(c, prop, objType, support, confidence);
  $\mathcal{S}_\delta$.add(s)
$\mathcal{S}_{n+1} \leftarrow \text{copy}(\mathcal{S}_n)$;
foreach *shape s in* $\mathcal{S}_\delta$ **do**
  $k \leftarrow \text{shapeKey}(s)$ ("$k = (c, p)$"); **if** $k \notin \mathcal{S}_{n+1}$ **then**
    $\mathcal{S}_{n+1}$.add(s)
  else
    $s_{\text{old}} \leftarrow \mathcal{S}_{n+1}[k]$;
    if *significance*$(s, \Psi^\delta_{\text{cec}}, \Psi^\delta_{\text{sts}}) >$ *significance*$(s_{\text{old}}, \Psi^n_{\text{cec}}, \Psi^n_{\text{sts}})$ **then**
      $\mathcal{S}_{n+1}$.replace(k, s)
    else
      markUnresolved(s)
$(\Psi^{n+1}_{\text{cec}}, \Psi^{n+1}_{\text{cpot}}, \Psi^{n+1}_{\text{sts}}) \leftarrow$ incUpdateMaps$(\Psi^n_{\text{cec}}, \Psi^n_{\text{cpot}}, \Psi^n_{\text{sts}}, \text{changes}, \text{deletions})$;
serialize$(\Psi^{n+1}_{\text{cec}}, \Psi^{n+1}_{\text{cpot}}, \Psi^{n+1}_{\text{sts}}, \mathcal{S}_{n+1})$

III. Evaluation

In this section, we present an evaluation of COSE and its effectiveness in performing continuous shapes extraction.

a) Software and Hardware Setup.: COSE is implemented in JAVA-11. The source code is available as open source [1] along with experimental settings and datasets. All experiments were conducted on a single machine with Ubuntu 22.04, equipped with an Intel(R) Core(TM) i9-10920X CPU @ 3.50GHz having 24 cores and 128 GB RAM. We compare COSE with state-of-the-art shape extraction tools: SheXer, QSE-FG (full-graph), and QSE-Approx (approximate variant). QSE-FG was run in exact mode ($\varepsilon = 0.0$), and QSE-Approx was executed with its default approximation threshold ($\varepsilon = 0.05$), as suggested in [4]. Default parameters were used for all tools unless otherwise stated. COSE implements both QSE-FG and QSE-Approx; unless explicitly specified, we run COSE with QSE-FG.

We invoke SheXer with the option `{output=SHACL` so that it produces SHACL-format shapes (instead of ShEx). This enables us to compare each tool's SHACL shapes directly against COSE's SHACL output, rather than comparing ShEx vs. SHACL. We also considered ShapeDesigner [11], and SHACLGEN [12]. However, as reported in prior work [4], their current implementations cannot handle KGs larger than a few million triples, and they do not manage to extract shapes of KGs having more than a few hundred classes. Since QSE-FG and QSE-Approx have been shown to outperform these tools in both scalability and shape quality [4], we excluded them from our experiments.

b) Datasets.: For our experiments, we selected multiple versions of three prominent real-world datasets containing hundreds of millions of triples: DBpedia, YAGO, and Wikidata. An overview of the content of these datasets is provided

[1] https://github.com/bayadhouib/COSE

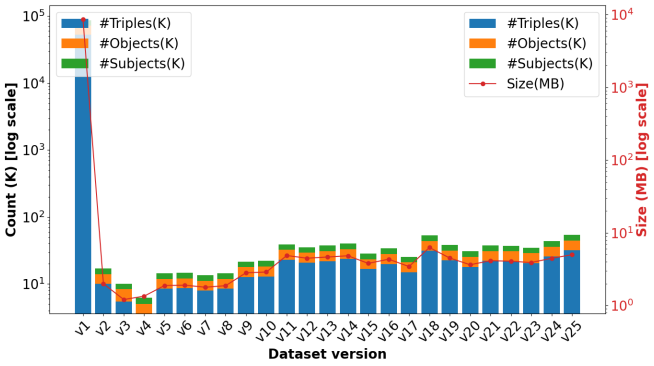


Fig. 2: Size and Statistics Across DBpedia Versions.

in table II and fig. 2. **DBpedia:** We used 25 consecutive versions starting from November 2014 and including each subsequent monthly release [13] [14].

YAGO: We used 4 versions: YAGO1, YAGO2, YAGO2S, and YAGO3. These versions were chosen to represent the major updates and enhancements that YAGO underwent, each combining information from Wikipedia and WordNet to various extents [15] [16] [17] [18].

Wikidata: We used 6 consecutive versions, starting with the latest-lexemes release, followed by several updates released between March and April 2024. [19] [20] [21]. Wikidata provides updates as incremental dumps, and each increment is around 150GB. However, SOTA shapes extraction tools, such as SheXer and QSE-FG, require the full graph for each new version and are unable to scale to such large graphs. To enable a fair comparison, we therefore generated a 23 GB subset from each version by uniformly sampling triples (fixed random seed = 42), which preserves the original (entity, property, object) distribution while keeping the graph tractable for all methods.

TABLE II: Size and Statistics of Datasets Versions.

Dataset version	# Triples	# Objects	# Subjects	Size
Wikidata-v1	938.59M	220.45M	133.31M	127GB
Wikidata-v2	168.41M	39.51M	23.90M	23GB
Wikidata-v3	168.66M	39.57M	23.94M	23GB
Wikidata-v4	168.89M	39.62M	23.98M	23GB
Wikidata-v5	168.99M	39.64M	23.99M	23GB
Wikidata-v6	169.53M	39.78M	24.08M	23GB
Yago1	18.26M	10.93M	2.22M	2.6G
Yago2	109.89M	46.03M	10.12M	16G
Yago2S	220.42M	38.67M	72.14M	37G
Yago3	121.23M	58.49M	15.17M	20G

A. Performance Analysis

In this section, we compare COSE's performance with SheXer, QSE-FG, and QSE-Approx using the three public KGs. Unlike COSE, SheXer, QSE-FG, and QSE-Approx only work on a single, complete snapshot of a graph. YAGO releases are full graph snapshots and hence can be directly used as input to all three baseline methods. Therefore, we applied the direct comparison approach to detect changes between consecutive YAGO versions. However, Wikidata and DBpedia include only updates in their releases. Thus, to run SheXer, QSE-FG, and QSE-Approx, we reconstructed the full graph for each version V_{n+1} of Wikidata and DBpedia by merging updates with the preceding version V_n.

TABLE III: Processing Time For Wikidata Dataset.

Dataset	SheXer	QSE-FG	QSE-Approx	COSE	Speedup
Wikidata-v1	42h	17h58min	9h15min	—	—
Wikidata-v2	48h	21h14min	10h55min	3h12min	6.6×–15×
Wikidata-v3	Timeout	24h30min	12h48min	3h50min	6.4×
Wikidata-v4	Timeout	27h47min	13h35min	4h35min	6.1×
Wikidata-v5	Timeout	31h05min	14h42min	4h51min	6.4×
Wikidata-v6	Timeout	34h23min	15h31min	5h09min	6.9×

TABLE IV: Processing Time For YAGO Dataset.

Dataset	SheXer	QSE-FG	QSE-Approx	COSE	Speedup
YAGO1	3h30min	48min	24min	—	—
YAGO2	14h	6h20min	3h09min	40min	9.5×–21×
YAGO2S	23h16min	12h38min	6h55min	1h35min	8×–14.7×
YAGO3	15h53min	7h21min	3h55min	58min	7.6×–16.4×

table III, table IV, and fig. 3 show the execution time of all four solutions under various datasets. First, we observe that COSE and QSE-Approx are able to handle large graph sizes. With Wikidata, SheXer fails to scale beyond v2, as it times out for larger graph sizes. QSE-Approx performs better than QSE-FG in all cases but still suffers from full-graph dependency. Second, COSE consistently outperforms all three baselines. It achieves a 6×–28× reduction in execution time across all datasets. This is because the execution time of SheXer, QSE-FG, and QSE-Approx is proportional to the size of the complete graph, whereas COSE's execution time is proportional to the delta graph size. This also explains the difference in speed-ups observed across datasets. For instance, as the Wikidata graph size doubles from v1 to v6, QSE-FG's execution time nearly doubles (17h to 34h), while COSE scales more efficiently. Similarly, datasets with smaller updates, like DBpedia, show a much larger performance gap, where COSE outperforms QSE-FG and SheXer by up to 10× and 28×, respectively.

fig. 4 presents a breakdown of COSE's processing time across the core stages for the three KGs (only v2 is shown, as the results with other versions are similar). First, we notice that only YAGO has extra time spent on the change detection stage. For Wikidata and DBpedia, new versions of the graph are released as updates, and COSE skips change detection. Second, the majority of the time is spent on the validation-based filtering optimization and delta graph generation. Generating SHACL shapes from the graph and performing shapes merging/conflict resolution is fast over the delta graph. We can also see that validation takes more time than graph generation for YAGO and DBpedia, while the converse is true for Wikidata. The difference in complexity of updates across these KGs results in variation across these stages, and the

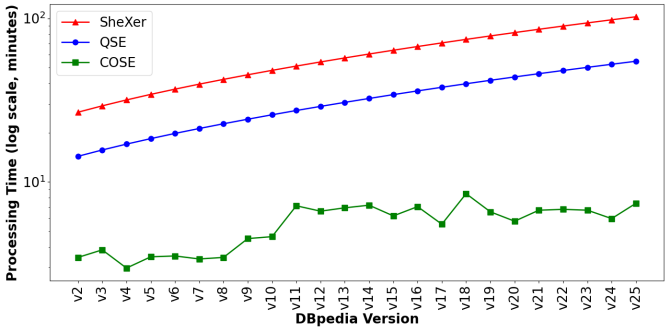


Fig. 3: Processing Time for DBpedia Dataset.

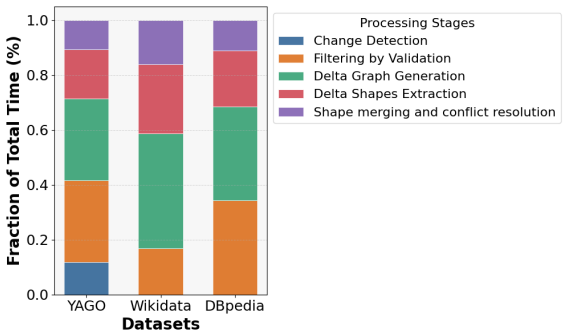


Fig. 4: Time Taken At Various Stages For Different Datasets.

resulting differences in the delta graph are the reason why COSE provides different degrees of improvement for the three KGs; while COSE provides up to 7× improvement with Wikidata, it provides up to 28×/21× improvement with DBpedia/YAGO respectively.

B. Accuracy Analysis

We now focus our evaluation on the accuracy of COSE. table V reports the number of SHACL shapes generated by COSE, QSE-FG, QSE-Approx, and SheXer for each of the three public datasets. We provide the results for v2 here. We present results for other versions later in Section III-B5 when we evaluate drift across increments. As can be seen, both QSE-FG and QSE-Approx, as well as COSE , extract more shapes than SheXer. This reproduces results from prior work [4] that demonstrated that QSE-FG extracts more relevant shapes than SheXer, and also shows that COSE and QSE-Approx can provide better accuracy than SheXer even when operating under incremental or approximate assumptions. While QSE-Approx slightly underperforms QSE-FG in terms of precision and recall, it remains competitive and demonstrates the tradeoff between scalability and exactness. Notably, COSE achieves accuracy comparable to QSE-Approx, despite working on delta graphs.

To understand the difference between COSE and QSE-FG, we used SHACL shapes generated by QSE-FG as the ground truth and computed precision and recall scores for COSE . Precision indicates the proportion of relevant shapes extracted compared to all shapes extracted by COSE. Recall measures

TABLE V: Accuracy Results Across Tools and Datasets (Shapes Extracted / Precision / Recall).

Tool	Wikidata-v2	YAGO2	DBpedia-v2
QSE-FG	47,512 / 100% / 100%	4,656 / 100% / 100%	1,372 / 100% / 100%
QSE-Approx	47,000 / 93.2% / 94.5%	4,610 / 92.3% / 93.1%	1,360 / 93.5% / 94.1%
SheXer	26,132 / 81.5% / 78.3%	2,328 / 80.4% / 76.9%	823 / 82.2% / 79.5%
COSE	46,850 / 92.5% / 91.8%	4,617 / 91.2% / 90.5%	1,362 / 93.5% / 92.8%

TABLE VI: Categorization Of COSE-generated SHACL Shapes With Respect To QSE-FG-generated SHACL.

Dataset	%Overlap	Conflicting (%)	Distinct (%)
		Total / Resolved / Unresolved	
Wikidata-v2	10	5 / 4.8 / 0.2	85
DBpedia-v2	30	20 / 18.8 / 1.2	50
YAGO2	22	20 / 18.7 / 1.3	58

the proportion of relevant shapes extracted by COSE out of all shapes extracted by QSE-FG. table V shows these values for various KGs. As can be seen, COSE provides a precision and recall of more than 90.5% across all KGs. This result raises four questions:

- Which SHACL shapes differ between QSE-FG and COSE leading to the drop in accuracy?
- Which stage of COSE's pipeline is responsible for the drop in accuracy?
- What is the effect of this accuracy loss when these SHACL shapes are used to validate the KG?,
- How does accuracy drop further as we perform incremental shapes extraction across several versions?

1) Analyzing shape constraints.: To answer these questions, we first analyzed the evolution of SHACL shapes from the base-KG V_n and the delta-KG V_δ. We categorized SHACL shapes generated by COSE for V_δ into one of three types: (i) distinct, (ii) overlapping, and (ii) conflicting. Overlap represents the case where COSE generated a shape from V_δ that is identical to one already generated from V_n. Conflict represents the case where COSE generates a shape from V_δ that corresponds to a shape generated from V_n, but the two shapes are different. Distinct represents the case where COSE generates shapes that are unique to KG_δ. table VI shows the breakdown of these categories across various datasets. As can be seen, the distinct category accounts for the largest proportion of shapes in all KGs. This suggests that incremental updates across the three KGs result in many more new SHACL shapes being generated compared to existing SHACL shapes being modified.

Figures 5a,5b, and 5c present an analysis of the type of updates across various versions of the three KGs. For each version, the figures show a breakdown of updates into three categories, namely new triples that have been added to the KG (Added), modifications to triples that already exist in the KG (Modified), and deletion of triples that exist in the KG (Deleted). As can be seen, across all KGs, updates are dominated by the addition of new triples. This explains the high proportion of distinctly new SHACL constraints, as they are derived by COSE from these new triples. Further analysis

TABLE VII: Distribution Of Detected Conflict Types.

Conflict Type	Wikidata v2	DBpedia v2	YAGO2
Class Violation	63%	49%	57%
Datatype Mismatch	37%	36%	43%
Cardinality Violation	0%	15%	0%

revealed that neither distinct nor overlap categories contributed to accuracy loss, as shapes generated by COSE in these categories were also found to be generated by QSE-FG. This points to the conflict category as the source of accuracy loss.

To further analyze the conflicting shapes, we categorized them into various types: (i) class violations, where entities are assigned to incorrect classes, thereby violating class-based constraints (for example `dbo:Place` is reclassified as `dbo:Location`), (ii) node kind violations, where IRIs are replaced with literals or vice versa, breaking expected node types (for example in Wikidata, an IRI such as `Q42` may be replaced by the literal `'Douglas Adams'` in a property that expects and IRI), (iii) property datatype violations, such as converting integers to strings, dates to incorrect formats, or using invalid values for certain fields (for example `dbo:populationTotal` transitions from `xsd:integer` to `xsd:string`), (iv) pattern violations that occur by deviating from predefined regular expressions, such as incorrectly formatted email addresses or URLs, (v) cardinality violations by exceeding or falling short of the allowed number of property occurrences (`sh:minCount` or `sh:maxCount`), and (vi) range violations, where values are assigned outside permissible boundaries, such as negative values where only positive numbers are allowed (for example `dbo:age` might incorrectly include a negative value). table VII shows the fraction of conflicting shapes across various categories. As can be seen, class violations and datatype mismatches are the most common across the three KGs.

Recall that the COSE pipeline generates SHACL shapes using the enriched delta graph and then performs SHACL merging and conflict resolution. There are two possible ways these violations could have been handled by COSE. The first case is where the SHACL merge stage took conflicting shapes and merged them successfully with shapes from V_n. The second case is where the merge process fails. table VI shows a breakdown of the conflict category across these two possibilities (Resolved and Unresolved). As can be seen, a majority of conflicts were automatically merged by COSE.

TABLE VIII: Breakdown Of Resolved And Unresolved In Various Categories Across Datasets.

Dataset	Category	Class Violation	Datatype Mismatch	Cardinality Violation
Wikidata v2	Resolved	48%	52%	–
	Unresolved	92%	8%	–
DBpedia v2	Resolved	42%	38%	20%
	Unresolved	78%	18%	4%
YAGO2	Resolved	50%	50%	–
	Unresolved	88%	12%	–

table VIII shows a breakdown of resolved and unresolved conflicts across various categories. As can be seen, class violations account for a majority of unresolved conflicts across all KGs. Recall that a class violation occurs when entities are assigned to different classes in the shapes extracted from base-KG and delta-KG. This suggests that the distribution of entity types created by sampling in delta KG differs from that of the base KG.

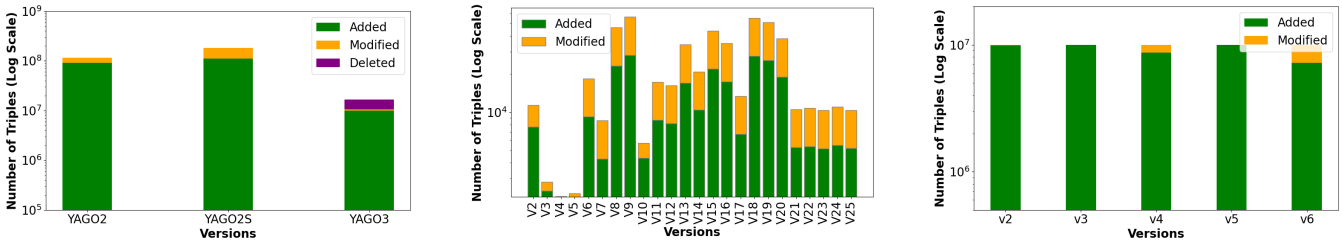
2) Impact of Sampling Strategies: To evaluate the effect of sampling, we compare Type-Specific Sampling (TSS) with two other strategies, namely, Full Sampling (FS) and No Enrichment (NS). In the FS case, COSE adds all entities of the same type (instead of just a random sample as done by TSS) to the delta graph. In the NS case, COSE completely skips sampling and uses only the changed triples as its input. The performance and accuracy results of the three strategies are summarized in table IX. First, comparing TSS with FS, we can see the performance–accuracy trade-off. When we perform full sampling, we see that COSE achieves 100% accuracy for YAGO and DBpedia, and 99.8% accuracy for Wikidata. Manual inspection of shapes that did not match the ground truth showed that the small discrepancy with Wikidata is due to schema changes introduced by updates that lead to minor variations in a few shapes. However, this high accuracy comes at the trade-off of performance, as FS is 7–21$\times$ slower than TSS.

Second, comparing TSS with NS, we can see that skipping enrichment completely provides 2$\times$ improvement in performance. However, this comes at the price of accuracy, as it can lead to up to 10% loss in precision and recall with the public KGs. We estimate that this fraction is optimistic because updates in these public KGs are mostly additions of new triples that lead to new SHACL shapes, as we showed earlier. In other scenarios where updates are modification intensive, a lack of sampling would lead to a much bigger loss in accuracy. TSS thus provides a middle-ground in balancing performance and accuracy, as it provides performance close to NS with accuracy close to FS.

TABLE IX: Impact Of Sampling Strategies On Precision And Processing Time.

Version	Sampling	Precision (%)	Recall (%)	Time
Wikidata v2	TSS	92.5	91.8	3h12min
	FS	99.8	99.5	(>, 24 hours)
	NS	81.9	78.2	1h46min
DBpedia v2	TSS	93.5	92.8	31min
	FS	100	100	3h30min
	NS	88.6	85.1	11min
YAGO2	TSS	91.2	90.5	40min
	FS	100	100	14h00min
	NS	84.3	80.3	16min

3) Sensitivity Analysis of Sampling Parameters: We now present a sensitivity analysis of COSE's sampling parameters ((α) and ($k_{\min}$). To recall, these parameters are used to compute the sampling threshold, where α controls the fraction of



(a) Detected Changes in YAGO. (b) Detected Changes in DBpedia. (c) Detected Changes in Wikidata.

Fig. 5: Detected Changes Across Different Knowledge Graphs

entities sampled from the base graph V_n and k_{min} ensures a minimum number of triples are included. In all experiments presented in this evaluation, we set $\alpha = 0.15$ and $k_{min} = 5000$. In contrast, table X shows results for varying α with k_{min} fixed at 5000 and table XI shows results for varying k_{min} with α fixed at 0.15. Across all datasets, we see the performance–accuracy tradeoff; increasing α or k_{min} provides an improvement in accuracy and a drop in performance, and decreasing both has the opposite effect. We set $\alpha = 0.15$ or $k_{min} = 5000$ for all experiments in this evaluation, as these values provided the best tradeoff between performance and accuracy. Beyond this value, we observed only marginal improvements in accuracy with a much bigger drop in performance. For instance, increasing α to 0.25 incurs a slowdown of up to 1.7×, while improving precision/recall only by up to 1.4%/0.5%.

TABLE X: Sensitivity Analysis Of α With $k_{min} = 5000$.

Dataset	α	Processing Time	Precision (%)	Recall (%)
3*DBpedia-v2	0.05	4min21	88.7	85.3
	0.15	6min09	93.5	92.8
	0.25	9min45	94.9	93.1
3*Wikidata-v2	0.05	2h23min	87.2	84.8
	0.15	3h12min	92.5	91.8
	0.25	5h24min	93.9	92.3
3*YAGO2	0.05	30min17	85.8	82.3
	0.15	40min	91.2	90.5
	0.25	1h09min	92.3	90.4

TABLE XI: Sensitivity Analysis Of k_{min} With $\alpha = 0.15$.

Dataset	k_{min}	Processing Time	Precision (%)	Recall (%)
3*DBpedia-v2	1000	4min97	89.8	86.4
	5000	6min09	93.5	92.8
	10,000	10min12	95.4	94.2
3*Wikidata-v2	1000	2h06min	88.7	85.2
	5000	3h12min	92.5	91.8
	10,000	5h47min	93.2	93.2
3*YAGO2	1000	23min11	87.6	84.1
	5000	40min	91.2	90.5
	10,000	1h18min	92	91.5

4) Practical Implication: Comparison of QSE-FG and COSE: To assess the practical implication of QSE-FG and COSE and to answer the third question raised earlier, we evaluate the correctness of their extracted SHACL shapes and their impact when used to validate the graph. We extract shapes from DBpedia v2 using both QSE-FG and COSE. Rather than selecting five arbitrary shapes, we employed a stratified sampling strategy to ensure statistical representativeness: we chose five SHACL shapes, each targeting a distinct class (i.e., `dbo:Village`, `dbo:Film`, `dbo:Organisation`, `dbo:Software`, and `dbo:City`). Each of these five shapes governs between 92 and 342 property constraints and collectively covers over 1,000 triples sampled from millions of entities, capturing both true positives and false negatives. To confirm that this five-shape sample was sufficiently stable, we repeated the selection process using two additional random seeds (17 and 91), observing a minimal variance of $\pm 1.2\%$ in precision and recall. These checks indicate that (1) each class exhibits a distinct constraint pattern, (2) each shape covers well over 100 governed triples meeting common thresholds for statistical significance and (3) the resulting precision/recall metrics converge, validating our sampling methodology. Precision and recall are calculated as follows: Precision: $\frac{TP}{TP+FP}$, Recall: $\frac{TP}{TP+FN}$, where TP (True Positives) are correctly identified shapes, FP (False Positives) are incorrect shapes, and FN (False Negatives) are valid shapes that were missed.

table XII shows the statistics, precision, and recall of property shapes extracted by QSE-FG and COSE across these classes. The total number of entities per class is reported in column Entities, while PS_ALL represents the total number of property shapes. Columns PS_Correct and PS_Wrong indicate how many extracted shapes were deemed valid or spurious after manual inspection. PS_QSE-FG and PS_COSE denote the number of shapes retained by QSE-FG and COSE, respectively, while the last four columns report true negatives, false negatives, precision, and recall for both methods.

As can be seen, QSE-FG achieves a perfect precision due its ability to effectively filter out spurious shapes while preserving valid ones using statistical confidence values derived from the full graph. COSE's 10% loss in accuracy actually results only in marginally lower precision values. This shows that COSE's validation is also effective despite working with the

TABLE XII: Analysis Of 5 Randomly Selected SHACL Shapes.

Class	#Entities	#PS_ALL	#PS Correct	#PS Wrong	#PS QSE-FG	True Neg QSE-FG	False Neg QSE-FG	Prec QSE-FG	Rec QSE-FG	#PS COSE	True Neg COSE	False Neg COSE	Prec COSE	Rec COSE
dbo:Village	310,248	342	74	98	72	98	2	1.0	0.97	69	93	5	0.96	0.93
dbo:Film	157,682	130	56	74	55	73	3	1.0	0.95	52	70	5	0.96	0.91
dbo:Organisation	46,218	122	45	77	44	75	2	1.0	0.96	41	71	4	0.95	0.91
dbo:Software	18,659	92	40	52	39	50	1	1.0	0.98	37	48	2	0.95	0.95
dbo:City	34,120	170	60	110	59	108	1	1.0	0.98	57	105	3	0.95	0.95

delta graph. Similarly, COSE retains marginally fewer shapes than QSE-FG across all classes we verified. For example, for the class `dbo:Village`, QSE-FG retains 72 shapes, while COSE retains 69. This explains QSE-FG's marginally higher recall, as by reprocessing the entire graph it preserves more valid constraints. In contrast, COSE's incremental approach occasionally omits a few shapes during updates, resulting in a small number of false negatives and a slight drop in recall. These results underscore an important trade-off. QSE-FG ensures a more comprehensive set of constraints but requires reprocessing the entire graph, making it computationally demanding for continuously evolving KGs. In contrast, COSE efficiently extracts a refined set of constraints by leveraging incremental updates, significantly reducing computational overhead ($6\times$–$10\times$) and making it advantageous for dynamic KGs that require frequent updates.

5) Drift Across Increments: We now present an analysis of COSE's accuracy as we use it across multiple KG versions to answer the fourth and final question raised earlier. Recall that COSE serializes and tracks merged shapes and data structures across versions. The question we aim to answer here is whether COSE's drop in accuracy compared to QSE-FG progressively deteriorates as we increase the number of versions.

fig. 6 and fig. 7 show the accuracy of COSE across all versions of three public KGs. As can be seen, COSE maintains a precision and recall of more than 90% across multiple versions in all three KGs; Wikidata and YAGO had less than a 0.5% drop in accuracy, while DBpedia had a 3% drop across 25 versions. This shows the robustness of COSE with the public KGs considered in this study. From our analysis of the practical implications of using COSE's shapes, we know that this drop in accuracy will not lead to a major drop in validation accuracy. However, we would like to explicitly point out here that a straightforward strategy of dealing with such drift across increments is to adopt a hybrid strategy for dealing with continuous shapes extraction. Similar to the way enterprise storage systems perform periodic full backup and interim incremental backups, COSE can be used to perform shapes extraction on a full KG periodically, with interim updates following incremental shapes extraction. The exact window between two full extractions can be customized depending on the update rate and update complexity of each graph. Recall that COSE tracks and serializes data structures and SHACL shapes across versions. Using this, COSE already supports such a hybrid shapes extraction strategy out of the box.

IV. Related Work

Early work on KG constraint extraction and validation focused on static data. Tools like NADEEF [22], LLUNATIC [23], and the unified repair model [24] process individual snapshots using fixed rules, offering no support for efficient updates and addressing data cleaning rather than shape extraction. QSE [4] scales SHACL shape extraction to large KGs but requires full recomputation for each version. SheXer [3] similarly targets static graphs, lacking reuse of prior shape computations. ShapeDesigner [11] and SHACLGEN [12] generate SHACL constraints but do not scale to large KGs and remain limited to single snapshots. These approaches operate on static snapshots and largely target data-cleaning or validation tasks, which are conceptually distinct from deriving SHACL constraints [25].

Incremental techniques for evolving data have been explored in related domains. IncRML [26] and Belhajjame et al. [27] incrementally maintain mappings or schema saturation but do not handle shape extraction. Volkovs et al. [28] address constraint adaptation for tabular data, not RDF. Fan et al. [5], [6] develop incremental graph query maintenance, while Papavasileiou et al. [29] and Onias et al. [30] target evolving query and ontology constraints. Bleifuß et al. [31] study temporal schema recommendations for user-curated web tables, providing complementary insights into schema evolution.Yet, none support continuous SHACL shape inference.

Other systems like Guided Data Repair (GDR) [32] rely on user feedback to fix violations but assume static constraints and perform validation per snapshot.

COSE bridges this gap by enabling continuous SHACL shape extraction over delta graphs, combining QSE's statistical inference [4] with incremental maintenance [5], [6], [27]. To our knowledge, it is the first end-to-end framework to support shape mining over evolving KGs.

V. Conclusion and Future Work

We introduced COSE, a system for continuous SHACL shape extraction over dynamic KGs. By adapting QSE to work incrementally over delta graphs, COSE avoids costly full graph reprocessing while maintaining shape quality. Experiments on real-world datasets show that COSE offers an order-of-magnitude performance gain over full-graph QSE with minimal accuracy loss.

Several future directions emerge. First, while type-specific sampling balances accuracy and performance, smarter strategies could further reduce the trade-off. Second, although COSE scales to large updates, real-time extraction remains a challenge; batching updates or lightweight filtering may offer

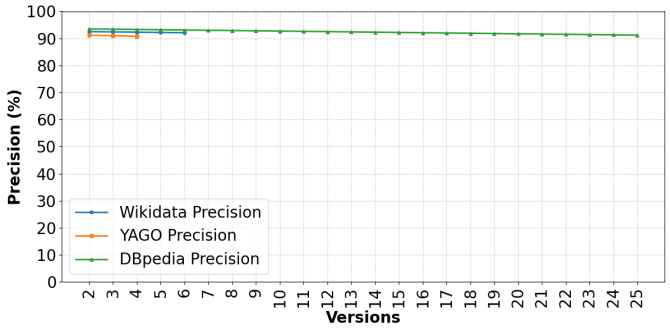


Fig. 6: Precision Of COSE Across Datasets.

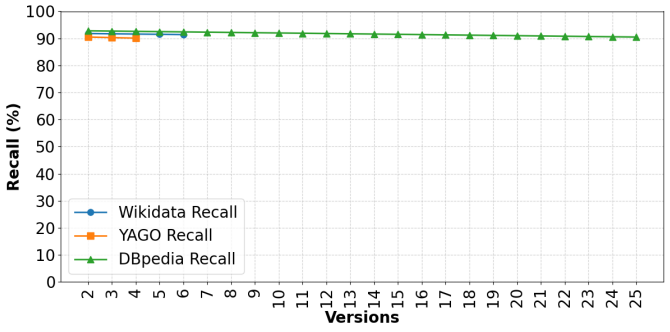


Fig. 7: Recall Of COSE Across Datasets.

practical solutions. Lastly, COSE assumes constraints persist across versions, but dynamic data may require evolving the constraints themselves. Our framework lays the groundwork to explore such adaptive validation in evolving KGs.

REFERENCES

[1] H. Knublauch and D. Kontokostas, "Shapes constraint language (shacl)," W3C Candidate Recommendation, 2017, available at: https://www.w3.org/TR/shacl/.

[2] E. Prud'hommeaux, J. E. Labra Gayo, and H. R. Solbrig, "Shape expressions: An rdf validation and transformation language," in *Proceedings of the 10th International Conference on Semantic Systems (SEMANTiCS)*. Leipzig, Germany: ACM, 2014, pp. 32–40.

[3] M. Poveda-Villalón *et al.*, "Automatic extraction of shapes using shexer," in *Proceedings of the ISWC 2021 Satellite Tracks (CEUR Workshop Proceedings)*. Springer, Cham, 2021, pp. 221–235.

[4] K. Rabbani, M. Lissandrini, and K. Hose, "Extraction of validating shapes from very large knowledge graphs," *Proceedings of the VLDB Endowment*, vol. 16, no. 5, pp. 1023–1032, 2023.

[5] W. Fan, X. Wang, and Y. Wu, "Incremental graph pattern matching," *ACM Transactions on Database Systems*, vol. 38, no. 3, pp. 12:1–12:47, 2013.

[6] W. Fan, C. Hu, and C. Tian, "Incremental graph computations: Doable and undoable," in *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*, 2017, pp. 155–169.

[7] SciencesDaily. (2020) Inconsistent data presentation on covid-19 dashboards risks public health efforts. Online. Retrieved from https://www.sciencedaily.com/releases/2020/08/200820110900.htm.

[8] BMJ Global Health. (2021) Challenges in composite data sourcing during the covid-19 pandemic. Online. BMJ Global Health, 6(5), e005542. Retrieved from https://gh.bmj.com/content/6/5/e005542.

[9] OpenStreetMap Community, "Best practices for mapping temporary road closures due to natural disasters," Online, 2020, retrieved from https://help.openstreetmap.org/questions/81647/best-practice-for-mapping-5-month-road-closure-with-approximate-end-date.

[10] E. Software, "Kryo: Fast, efficient java serialization," https://github.com/EsotericSoftware/kryo, 2024.

[11] I. Boneva, J. Dusart, D. Fernández-Álvarez, and J. E. Labra Gayo, "Shape designer for shex and shacl constraints," in *Proceedings of the ISWC 2019 Satellite Tracks (CEUR Workshop Proceedings)*, vol. 2456. Auckland, New Zealand: CEUR-WS.org, 2019, pp. 269–272.

[12] A. Keely. (2022) Shaclgen. Online. Retrieved from https://pypi.org/project/shaclgen/; Accessed 20th January.

[13] J. Lehmann, L. Bühmann, P. Westphal, S. Bin, M. Brümmer, C. Dirschl, and C. Stadler, "Dbpedia – a large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web*, vol. 7, no. 1, pp. 1–19, 2016.

[14] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, and C. Bizer, "Dbpedia – a large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015.

[15] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: A core of semantic knowledge," in *Proceedings of the 16th International Conference on World Wide Web*. ACM, 2007, pp. 697–706.

[16] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum, "Yago2: A spatially and temporally enhanced knowledge base from wikipedia," vol. 194, 2013, pp. 28–61.

[17] ——, "Yago2s: A spatially enhanced knowledge base from wikipedia and wordnet," vol. 194, 2013.

[18] F. Mahdisoltani, J. Biega, and F. M. Suchanek, "Yago3: A knowledge base from multilingual wikipedias," in *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[19] Wikimedia Foundation. (2014) Introducing wikidata to the linked data web. In *The Semantic Web – ISWC 2014*.

[20] ——. (2023) Wikidata: A free knowledge base. Retrieved October 10, 2024. [Online]. Available: https://www.wikidata.org/

[21] ——. (2024) Wikidata: A free knowledge base. [Online]. Available: https://www.wikidata.org/

[22] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang, "Nadeef: A commodity data cleaning system," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2013, pp. 541–552.

[23] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Llunatic: A local logic for universal constraints with attributes," in *Proceedings of the VLDB Endowment*, vol. 5, no. 11, 2012, pp. 1338–1349.

[24] F. Chiang and R. J. Miller, "A unified model for data and constraint repair," in *Proceedings of the IEEE 24th International Conference on Data Engineering (ICDE)*. Cancun, Mexico: IEEE, 2008, pp. 446–455.

[25] A. Polleres, A. Bonifati, O. Hartig, A. Hogan, M. Höller, S. Kirrane, M. Šimkus, S. Steyskal, and A. Zimmermann, "Rdf constraints: A systematic analysis and framework," *Journal of Web Semantics*, vol. 78, p. 100772, 2023.

[26] D. Van Assche *et al.*, "Incrml: Incremental knowledge graph construction from heterogeneous data sources," in *ESWC 2021: The Semantic Web – ESWC 2021 Satellite Events*. Springer, Cham, 2021, pp. 113–118.

[27] K. Belhajjame and M.-Y. Mejri, "Online maintenance of evolving knowledge graphs with rdfs-based saturation and why-provenance support," *Journal of Web Semantics*, vol. 64, p. 100584, 2020.

[28] M. Volkovs, C. Cheng, K. Doka, and G. Hinton, "Cleaner: Continuous learning for data repair," pp. 5787–5796, 2019.

[29] V. Papavasileiou, N. Konstantinou, Y. Tzitzikas, and D. Spanos, "Incremental query rewriting and processing in evolving rdf knowledge bases," *Journal of Web Semantics*, vol. 19, pp. 1–16, 2014.

[30] N. Onias *et al.*, "Maintaining ontology-based constraints in dynamic rdf datasets," *International Journal on Semantic Web and Information Systems*, vol. 11, no. 3, pp. 21–39, 2015.

[31] D. Bleifuß, R. Le Bras, I. F. Ilyas, and T. Rekatsinas, "Temporal schema recommendation for evolving web tables," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2025.

[32] M. Yakout, A. Elmagarmid, M. Ouzzani, and I. F. Ilyas, "Guided data repair," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 193–204, 2013.