Quality shapes extraction from Knowledge Graphs on Azure serverless functions

Francesco Dente, Michele Ferrero

1 Introduction

Knowledge Graphs (KGs), which are stored as collections of triples of the form *(subject, relation, object)* using the Resource Description Framework (RDF), are widely used both in the enterprise and on the Web. However, as KGs rapidly accumulate more data, practical applications impose further requirements for quality assessment and validation. Therefore, shape constraint languages, such as SHACL and ShEx, have been proposed to validate KGs by enforcing constraints represented in the form of validating shapes. For example, we can express that an entity of type student requires a name, a registration number, and should be enrolled in some courses; and that these attributes should be of type string, integer, and course, respectively. Often, validating forms are manually specified by domain experts. However, when trying to specify validating shapes for already existing large KGs, data scientists need tools that can speed up this process. Several tools have been proposed to automatically or semi-automatically generate a set of validating shapes for a target KG. We propose to start from the **QSE-Exact** algorithm described in the article **Extraction of Validating Shapes from Very Large Knowledge Graphs** [11] and run it on *Azure serverless functions*. In this way, we aim to make shape extraction **faster**, more **fault-tolerant**, and easier to run **periodically** on-demand thanks to the Azure serverless environment.

2 Preliminaries

We now briefly describe the key Azure cloud technologies that we leverage in our solution.

2.1 Azure function

Azure Function[6] is a serverless compute service that allows users to run event-triggered code without having to deploy or manage infrastructure. As a trigger-based service, it runs a script or piece of code in response to a variety of events. Azure functions can be used to achieve decoupling, high throughput, reusability, and sharing. Because it is more reliable, it can also be used for production environments.

Serverless functions are actually pay-as-you-go (no charge for idle time between calls), and they allow the provider to share a pool of servers among customers, so any worker can run any handler, there is no spinup time, and there is less switching. This comes at the cost of encouraging the use of a particular runtime (Java, C#, JavaScript, Python) and forcing the writing of stateless functions, which, as we will see, is one of the main problems we need to address in our solution.

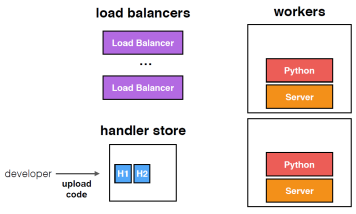

Figure 1: (a)

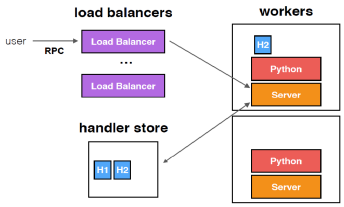

Figure 2: (b)

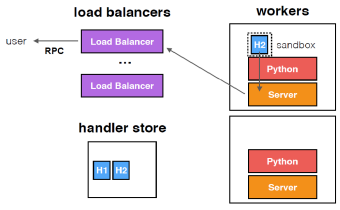

Figure 3: (c)

Figure 4: From top to bottom: (a) Developer uploads code on handler store; (b) Load Balancer schedules the function on a worker; (c) Response is sent back to the user when the function finishes;

2.1.1 Azure durable functions

To orchestrate the flow of the functions, we use **Azure Durable Functions**[5], a feature of Azure Functions that lets us write stateful functions in a serverless compute environment. The extension lets us define stateful workflows by writing orchestrator functions, and stateful entities by writing entity functions using the Azure Functions programming model. Behind the scenes, the extension manages state, checkpoints, and restarts for us. Using Durable Functions, we can then implement typical application patterns such as **Function Chaining** and **Fan Out/Fan In** that are fundamental to our algorithm.

Figure 5: Function chaining pattern

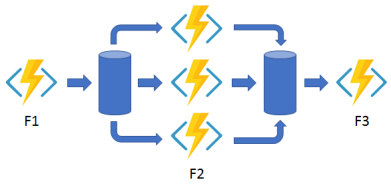

Figure 6: Fan In/Fan out pattern

2.2 Azure Blob Storage

Azure Blob Storage[7] is Microsoft's object storage solution for the cloud. Blob Storage is optimized for storing massive amounts of unstructured data. Unstructured data is data that doesn't conform to a specific data model or definition, such as text or binary data. Azure Blob Storage is heavily used by the Azure Function service, and we will also use it to pass large data structures as parameters between our serverless functions.

2.3 Azure Cache for Redis

Azure Cache for Redis[4] provides an in-memory data store based on Redis software. Redis improves the performance and scalability of an application that makes heavy use of back-end data stores. It's able to handle large volumes of application requests by keeping frequently accessed data in server memory, where it can be quickly written to and read from. Redis provides a critical low-latency, high-throughput data storage solution for modern applications. We will use Redis as a database to share data structures between our serverless functions.

3 Azure serverless QSE design

We try to keep the algorithm as close to the original as possible by taking the original phases and orchestrating them on the Azure functions in a Map Reduce fashion, using blob storage and cache for Redis to store intermediate results,

with further modifications to fit the serverless environment.

We first describe the main libraries and data structures used, then detail each phase of our algorithm, how it is implemented in our solution, and the key issues that need to be addressed.

Please refer to the original paper[11] for further explanation on the details of all the different data structures described in this report.

3.1 Libraries

3.1.1 Caffeine

Caffeine[1] is a high-performance, in-memory caching library optimized for fast and efficient key-value cache operations. It features intelligent memory management through various eviction strategies such as LRU (Least Recently Used) and LFU (Least Frequently Used). Designed for high performance, Caffeine ensures rapid read and write operations, making it ideal for performance-critical applications.

3.1.2 Kryo

Kryo[2] is a Java binary serialization framework renowned for its high-speed serialization capabilities and compact serialization results. By leveraging the ASM library for bytecode generation, Kryo ensures rapid execution. Compared to a well-known library like Protobuf, Kryo offers faster performance and better memory utilization, even though it lacks multi-language support. For this specific use case, we prioritized speed and efficient storage over multi-language compatibility.

3.1.3 Jedis

Jedis[3] is a lightweight Java library designed for efficient Redis integration, offering simplicity and speed compared to alternatives like Redisson. While it may lack some advanced features, for our specific needs, these were unnecessary trade-offs against performance. Key to our implementation was leveraging JedisPool, which effectively manages connections between our application and Redis.

3.2 Data structures

We now describe how we handle the main critical data structures of the original QSE-Exact algorithm.

3.2.1 ETD-EntityDataHashMap

The ETD data structure, as noted in the original paper [11], is the most critical in terms of memory usage. In addition, during phase 2 of our algorithm, each worker must have access to the entire ETD structure, regardless of how we partition our files. This is because the *object* entity of a triplet may not belong to the set of entities assigned to a worker according to the entity-based partitioning method described in section 3.3.1.

The full ETD structure cannot fit in the memory of a single function even for a medium-size graph, so we have to resort to Azure Cache for Redis as a storage medium to keep a central shared version of it.

We use redis hashmaps along with hget and hset to manage the ETD hashmap on redis.

To better adapt to the memory available on each function we add a write-back caching mechanism inside the data structure.

3.2.2 StringEncoder

In the original QSE-Exact, strings representing *property* and *object* are encoded to integer using the StringEncoder structure to reduce memory usage, which is one of the critical aspects of the algorithm. This structure needs to be shared by all workers, and it grows rapidly as the knowledge graph grows in size, so we cannot afford to keep a copy of it in each function, and we need to resort to Azure Cache for Redis to keep a central shared and consistent version of it.

Race conditions on the StringEncoder can't be avoided because it is incrementally populated by the different workers during each phase. To

solve this problem, we use lua scripts that are executed atomically on the Redis server.

To better adapt to the memory available on each function, we add a caching mechanism inside the data structure.

3.3 Phase 0 - File partitioning

3.3.1 Entity based partitioning

We want each of our workers[1] to be able to work on a chunk of the entire graph. However, we cannot split the graph into N chunks without any semantics, because then we would have to worry about race conditions on the shared Ψ_{ETD} between workers (see section 3.2.1 for further explanation on how the shared Ψ_{ETD} is managed on Redis).

Example problems:

- During **EntityExtraction**, if two different workers encounter the same entity associated with two different types, we must manage the race condition on the entry associated with the entity inside the shared Ψ_{ETD}.

- During **EntityConstraintsExtraction** the same thing could happen regarding properties of the same subject processed by two different workers.

Race conditions slow down the execution and are not trivial to handle using Redis, so we propose a **entity-based** partitioning of the file to solve this problem: each worker (function) is assigned a file containing only triplets where the subject is part of the set of entities assigned to it[2]. We create these entity-based partitions so that each worker works on roughly the same amount of data.

3.3.2 Implementation

Since the Azure blob storage API does not allow a function to read a blob line by line without first downloading the whole file, we need to first generate smaller blobs (subfiles) already correctly partitioned, so that each worker will not need to have the whole initial graph in memory when performing the subsequent phases.

This partitioning can't be executed by a single azure function due to its memory limitations[3], therefore, in order to keep everything on the cloud, an Azure virtual machine is used.

The partitioning of the initial graph into entity-based subfiles, although necessary to optimize the workload distribution among Azure functions, is a time-consuming process. However, it's important to note that this partitioning does not need to be performed each time the system is run. Once the initial graph is segmented into smaller, manageable subfiles based on entity assignments, these partitions can be reused in subsequent runs, and updates or modifications to the graph can be managed directly within these existing subfiles.

Algorithm 1 Partitioning on virtual machine

```
Read graph blob
subfiles ← Generate subfiles
for subfile in subfiles do
    Upload subfile in blob storage
end for
Trigger start of orchestration
```

3.4 Phase 1 - Entity Extraction

In the entity extraction phase, each worker reads its assigned subfile and parses each (*subject*, *property*, *object*) triple containing a type declaration (e.g, rdf:type) and for each entity it stores the set of its entity types and the global count of their frequencies, i.e., the number of instances for

[1] instance of an azure function

[2] e.g. Worker 1 is assigned with the set of entities Alice, Luca, Michele, so its partition will contain only triplets with Alice, Luca, or Michele as subject

[3] Each instance has a limit of 1.5GB of RAM

each class in maps Ψ_{ETD} (Entity-to-Data) and Ψ_{CEC} (Class-to-Entity-Count), respectively.

3.4.1 Implementation

To speed up the computation and to avoid flooding Redis with small requests[4], we work on batches of triplets, using the Redis pipelining mechanism[8].

Algorithm 2 Entity extraction - Worker i

```
1: for batch in subfile_i do
2:     nodes ← filterTypePredicate(batch)
3:     subjs ← parseSubjects(nodes)
4:     objs ← parseObjects(nodes)
5:     updateStringEncoder(objs)
6:     updateLocETD(subjs, objs)
7:     updateCEC(objs)
8: end for
9: Write Ψ_CEC shard on blob storage
10: Push Ψ_ETD local content on Redis
```

3.5 Phase 2 - Entity Constraints Extraction and Support computation

This phase includes phase 2 and part of phase 3 of the original algorithm.

Each worker performs a second pass on its assigned subfile to collect the constraints and metadata needed to compute the support and confidence of each candidate property form. Specifically, it parses all triples except triples containing type declarations (which can now be skipped) to obtain for each predicate the subject and object types from the map Ψ_{ETD} populated in the previous step. During the constraints extraction, a local version of Ψ_{CTP} (class-to-property) is populated, which is then used in the last phase to perform the shapes extraction.

Then, for each triplet within the associated subfile, a first local version of Ψ_{SUPP} is computed.

3.5.1 Implementation

Again, to speed up the computation and avoid flooding Redis with small requests, we work on batches of triplets using the Redis pipelining mechanism[8]. In addition, for optimization purposes, before starting the actual constraint extraction, all entries of Ψ_{ETD} that contain entities associated with the corresponding partition of the current worker are downloaded from Redis to the local cache[5]. It's important to note that we no longer need to push the local contents of Ψ_{ETD} to Redis as we did in EntityExtraction, since only Ψ_{SUPP}, Ψ_{CTP}, Ψ_{CEC} will be used for the shapes extraction.

Algorithm 3 Entity constraints extraction - Worker i

```
1: Cache Ψ_ETD entries of entities associated to subfile_i
2: for batch in subfile_i do
3:     subjs ← parseSubjects(batch)
4:     objTypes ← parseObjectTypes(batch)
5:     props ← parseProperties
6:     Cache Ψ_ETD entries associated to objects in batch
7:     updateStringEncoder(props)
8:     updateLocETD(subjs, objTypes, props)
9:     updateCTP(subjs, objTypes, props)
10: end for
11: Compute local Ψ_SUPP
12: Write Ψ_CTP, Ψ_SUPP shards on blob storage
```

3.6 Phase 2.5 - Merging data structures

In this phase, all the shards of Ψ_{SUPP}, Ψ_{CTP}, Ψ_{CEC} created in the previous phases are merged.

3.6.1 Implementation

The merge is performed by three different Azure functions (one for each data model) running in

[4]Requests to Redis are at the entity level, so for large graphs there could be a huge number of them

[5]To make each worker aware of the entities it is assigned to, each subfile contains the complete list of entities in the first line

parallel. These data structures are not memory intensive, even for larger graphs, so this operation can be performed by a single Azure function.

Algorithm 4 Merging - Orchestrator

```
Schedule mergeSUPP
Schedule mergeCTP
Schedule mergeCEC
Wait for all functions to finish
```

3.7 Phase 3 - Confidence Computation and Shapes Extraction

This phase includes phase 4 and part of phase 3 of the original algorithm. Using information inside the merged Ψ_{SUPP} and Ψ_{CEC} we compute Ψ_{CONF}. Finally, in the shapes extraction phase, the algorithm iterates over the values of the Ψ_{CTP} map and defines the shape name, the shape's target definition, and the set of shape constraints for each candidate class.

3.7.1 Implementation

For the reasons explained in Section 3.6.1 we can use a single Azure Function to perform these computations.

Algorithm 5 Shapes Extraction

```
Compute Ψ_CONF
for entry in Ψ_CTP do
    buildShapes(entry)
end for
for entry in Ψ_CTP do
    prunedShapes(entry, Ψ_SUPP, Ψ_CONF)
end for
```

4 Evaluation

In this section we evaluate our QSE solution and compare it with the original single-thread **QSE-exact** algorithm.

4.1 Experimental setup

We perform benchmarks on two datasets that are part of the CIMPLE[9] project[6] (CimpleJanvier, CimpleAvril), a synthetic dataset generated using the lubm generator[10], and a sample of Wikidata.

The algorithm is implemented in **Java 17**, and experiments are conducted on a single machine running **Ubuntu 22.04** with **16 GB of memory** and a **AMD Ryzen 7 4800H 2.9 GHz CPU** with **8 cores and 16 threads**. All of the Azure infrastructure and the redis server are also running on the same machine.

	Janvier	Avril	Lubm	Wikidata
# of triplets	111k	486k	6.89M	34.54M
# of entities	20.56k	26.47k	1.08M	4.86M
Size	24 MB	110 MB	1 GB	4.9 GB

Table 1: Size and characteristics of datasets

4.2 Benchmarks

We now present the runtime metrics for different algorithms across various datasets. Additionally, we evaluate a version of our algorithm without the **StringEncoder**, accepting a trade-off of higher memory consumption for quicker runtime. We limit our function count to 1, 10, and 20 due to the constraints of our 16-thread machine; using more functions wouldn't result in true parallelism but would instead increase overhead. The data presented are averages from 10 runs under both warm and cold start conditions of the Azure environment. Notably, we exclude the time taken to split the initial graph into subfiles from these measurements.

[6]They contain information from fact-checking organizations

Model	1 subset	10 subsets	20 subsets
Warm runs			
No StrEnc	11.90s	2.50s	2.48s
StrEnc	18.80s	5.63s	6.08s
Single Thread	3.30s	-	-
Cold runs			
No StrEnc	13.20s	4.30s	4.60s
StrEnc	19.20s	5.50s	7.05s
Single Thread	3.30s	-	-

Table 2: Benchmarks on **CimpleJanvier**

Model	1 subset	10 subsets	20 subsets
Warm runs			
No StrEnc	29.00s	5.60s	5.63s
StrEnc	77.10s	19.60s	18.60s
Single Thread	3.80s	-	-
Cold runs			
No StrEnc	30.50s	9.13s	7.22s
StrEnc	77.80s	20.23s	20.12s
Single Thread	3.80s	-	-

Table 3: Benchmarks on **CimpleAvril**

Model	Runtime
No StrEnc	Out of memory
StrEnc	Out of memory
Single Thread	77.20s

Table 4: Benchmarks on **WikidataSample**

Model	1 subset	10 subsets	20 subsets
Warm runs			
No StrEnc	478.20s	166.7s	191.32s
StrEnc	1265.20s	281.24s	282.49
Single Thread	23.50s	-	-
Cold runs			
No StrEnc	480.06s	170.10s	189.20s
StrEnc	1270.65s	287.03s	289.30
Single Thread	23.50s	-	-

Table 5: Benchmarks on **lubm-big**

StringEncoder vs NoStringEncoder: The NoStringEncoder variant operates faster, at the cost of increased memory usage. This performance difference is evident in the CimpleAvril and lubm-big datasets, where managing race conditions on the StringEncoder and maintaining it as a central shared structure in Redis (despite caching mechanisms) significantly slows execution.

Both solutions cause an out of memory error when running on a large dataset like Wikidata, although the one with StringEncoder should not. This happens because the redis server runs on the same local machine as the QSE, and this greatly increases the RAM usage, which is only 16 GB. This is to be expected, since even the single-threaded algorithm sometimes has its process killed by the OS due to high memory usage if the RAM of the local machine is not almost completely available.

Comparison with single-thread QSE-exact: We now compare the performance of the cloud-based algorithms with the single-thread QSE-exact algorithm. It's important to note that the benchmarks for the cloud algorithms do not include the time taken to split files, as this is assumed to be a one-time operation with subsequent updates being less computationally demanding.

We can see in table 2 that the **NoStringEncoder** solution is faster than the **single-thread QSE** as the number of subsets increases. This is a promising result, although the same is not true for the other bigger datasets.

We believe that further increasing the number of subsets would lead our algorithm to perform better than the original one also for the other datasets, but we cannot test this on our local machine since we cannot have real parallelism with more than 20 functions.

Conclusions: Encouraging results were achieved predominantly with the smaller CimpleJanvier dataset. However, it is important to note in addition to the previous observations, that all experiments were conducted in an emulated Azure environment on our local machine, which may have added additional execution overhead.

Dataset	CimpleJ	CimpleA	lubm-big
EntityExtraction			
ETD Calls	20560	85762	1382214
ETD Misses	20560	26471	1083816
EntityConstraintsExtraction			
ETD Calls	334472	1343615	20630938
ETD Misses	31998	146350	1383214

Table 6: Redis ETD cache calls/misses with 1 function and NoStringEncoder model without pipelining

Dataset	CimpleJ	CimpleA	lubm-big
EntityExtraction			
ETD Hits	0	0	0
ETD Misses	20560	26471	1083816
ETD Jedis Calls	20560	26471	1083816
StrEnc Hits	18678	83882	1380265
StrEnc Misses	0	0	0
StrEnc Jedis Calls	119	491	6897
EntityConstraintsExtraction			
ETD Hits	66051	302657	3010371
ETD Misses	108189	467376	7936055
ETD Jedis Calls	103019	465036	6522664
StrEnc Hits	318131	1419322	17656040
StrEnc Misses	0	0	0
StrEnc Jedis Calls	149	521	6917

Table 7: Redis ETD and StringEncoder cache calls/misses with 10 functions with pipelining. We also report Jedis calls, we consider a pipeline of operations as a single call, that's why the ETD misses are greater than the calls to Jedis.

5 Conclusion

In this report, we presented an adaptation of the QSE-Exact algorithm for extracting shapes from Knowledge Graphs, utilizing Azure serverless functions to enhance scalability and fault tolerance. Our experimental results highlight both the benefits and limitations of the serverless approach. While the serverless model demonstrated promising speedups with smaller datasets, it encountered challenges when applied

to larger graphs. These challenges were primarily due to memory constraints, the overhead associated with managing shared data structures in a distributed environment, and the limited parallelism of our test machine.

Looking ahead, one possible improvement involves addressing high memory usage by changing the runtime environment. Currently, our implementation uses Java, which may not efficiently handle memory due to its garbage collection system. A potential alternative is switching to a runtime like Rust, known for its memory efficiency. However, making such a switch would be complex, as Rust is not currently supported by Azure Durable Functions and would necessitate significant adaptations to our existing codebase. This change could potentially mitigate some of the memory-related challenges highlighted in our experiments with larger datasets.

6 Additional notes

Currently the split in subfiles must be performed manually before running the algorithm and it's not yet integrated in the whole algorithm flow. There is a java file called RDFProcessor inside utils that performs a very naive non-parallel entity-based splitting.

The code we refer to in the report is the one insde the branch "dev-pipelines-new" in the gitlab repository, which contains a README with detailed instructions on how to run the code.

References

[1] Caffeine library.

[2] Kryo library.

[3] Jedis library.

[4] Azure cache for redis.

[5] Microsoft Azure Durable Functions.

[6] Microsoft Azure Functions.

[7] Azure blob storage.

[8] Redis pipelining.

[9] CIMPLE project.

[10] Guo, Yuanbo, Pan, Zhengxiang, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. pages 158–182, 2005.

[11] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. Extraction of validating shapes from very large knowledge graphs. *Proc. VLDB Endow.*, 16(5):1023–1032, 2023.