Materialisation approaches for Façade-based data access with SPARQL

Luigi Asprino [a,*], Enrico Daga [b], Justin Dowdy [c], Aldo Gangemi [d] and Paul Mulholland [b]
[a] *University of Bologna, Italy*
E-mail: luigi.asprino@unibo.it
[b] *The Open University, United Kingdom*
E-mails: enrico.daga@open.ac.uk, paul.mulholland@open.ac.uk
[c] *Semantic Arts Inc., CO, United States*
E-mail: justin.dowdy@semanticarts.com
[d] *Istituto di Scienze e Tecnologie della Cognizione, Consiglio Nazionale delle Ricerche, Italy*
E-mail: aldo.gangemi@cnr.it

Abstract. The Knowledge Graph concept is gaining momentum as an ideal approach to data integration. Therefore, it is of paramount importance to equip knowledge engineers with tools for accessing data from multiple, heterogeneous resources. The successful W3C standard SPARQL is the reference language for interacting with RDF knowledge graphs. For that reason, approaches extend SPARQL for accessing data in non-RDF formats. Recent research proposes relying on an intermediate RDF model, named Façade-X , whose components can be transparently mapped to various file formats. However, although Façade-X specifies how its components map to many different formats (CSV, JSON, HTML, Markdown, and others), it is still unclear how to implement a SPARQL execution engine that relies on it. In other words, what are the possible strategies for executing Façade-X queries? This article explores materialisation approaches for executing Façade-X queries. Specifically, we study two *in-memory* strategies for performing Façade-X data access with SPARQL. A *complete materialised view* strategy fully transforms the data source into RDF. Instead, a *sliced materialised view* strategy segments the data source and generates an RDF view on each part. Both strategies can be optimised by only materialising the part of the RDF graph that has potential matches with triple patterns in the query (triple-filtering). In addition, we compare these approaches with an *on-disk* alternative, which relies on a temporary database instance. We analyse the characteristics of these methods and perform extensive experiments, reporting on benefits and limitations of both approaches. Finally, we contribute guidelines and best practices derived from the findings.

Keywords: Knowledge Graph Construction, Data Integration, Facade-X

1. Introduction

The Knowledge Graph concept is gaining momentum as an ideal approach to data integration. Therefore, it is of paramount importance to equip knowledge engineers with tools for accessing data from multiple, heterogeneous resources. The W3C standard SPARQL 1.1 [18] is the reference language for interacting with RDF knowledge graphs. Similarly to SQL for relational databases, SPARQL has methods for selecting and filtering data, and for projecting them into tabular form. However, differently from SQL, SPARQL can also project the result into an RDF template, using the CONSTRUCT query type. Relying on this native feature, current research explored the application of SPARQL to a range of use cases broader than querying, specifically, the integration of heterogeneous data sources into knowledge graphs [13, 21, 22].

Recent research [8, 15] proposes to rely on an intermediate RDF model, named Façade-X, whose components can be transparently mapped to various file formats. This method allows to build software that pro-

*Corresponding author. E-mail: luigi.asprino@unibo.it.

vides *indirect access* to source data as-RDF, relieving knowledge engineers from the task of dealing with the variety of formats and related languages they rely upon – *re-engineering* (i.e. transforming resources by minimising domain considerations and focusing on the syntactical meta-model), and letting them focus on the semantic lifting – *remodelling* (i.e. re-framing the original domain model into a new one) [15].

Façade-X enables uniform access to a wide range of data formats *as-if* they were RDF, including the popular CSV, JSON, HTML, and XML and it was successfully applied to many complex scenarios, from scraping the content of Web sites, joining data from multiple sources [12], and building knowledge graphs from MusicXML scores [24]. Crucially, it was demonstrated how Façade-X can in principle represent any format expressed in a BNF grammar, as well as the relational data model [8]. However, although Façade-X specifies how its components map to these formats [15], it is still unclear how to implement a SPARQL execution engine that relies on it. In other words, which are the possible strategies for executing Façade-X queries?

In this article, we explore for the first time this research question and focus on materialisation strategies, that is generating the intermediate Façade-X RDF dataset in-memory, and then executing the SPARQL query (in contrast, for example, to performing query-rewriting). We elaborate two variants of this method for executing Façade-X queries *in-memory*. A *complete materialised view* strategy performs a full transformation of the data source into an in-memory RDF dataset. This approach has the benefit of providing full querying capabilities on the whole data but it can be problematic when the generated dataset requires more memory than what is available. To overcome this problem, we propose a *sliced materialised view* strategy, based on the assumption that data sources are collections of isomorphic items (CSV rows, JSON arrays,...). In this alternative approach, we segment the data source *before* transforming it into RDF, and execute the query separately on each one of the partitions. In addition, both strategies can be optimised by materialising into RDF only the data points that have potential matches with the basic graph pattern in the query. Therefore, we study the impact of a *triple-filtering* component on both *in-memory* execution strategies. Finally, we complement in-memory approaches with an alternative where the intermediate, Façade-X -based knowledge graph is materialised in a temporary database (*on-disk* strategy). To evaluate the benefits and costs of these approaches, we extend the GTFS-Madrid benchmark [10], and implement its queries as Façade-X based data access queries, compliant with the specification provided in [15]. We consider two types of data sources: CSV and JSON. This allows us to observe the behaviour of the strategies in two different data types, tabular and hierarchical, therefore, giving us the opportunity to make considerations that are independent from the details of the low-level, format-specific parser. We perform extensive experiments with four modalities: *in-memory+complete*, *in-memory+triple-filtering*, *sliced+triple-filtering*, and *on-disk+triple-filtering*. Crucially, we report on benefits and limitations of the various approaches in terms of performance (time) and memory requirements. Specifically, we contribute:

1. An analysis of execution strategies for performing Façade-X queries via materialisation
2. A *triple-filtering* optimisation method
3. An extension of the GTFS benchmark for evaluating execution engines of Façade-X queries
4. Guidelines and recommendations based on the findings of experiments' results

The rest of the article is structured as follows. The next section provides an introduction to façade-based data access, in conformance with [15]. Section 3 provides the details of our approach, and related hypotheses. Section 4 presents the experimental framework and discusses our findings. Related work is addressed in Section 5, before concluding the paper in Section 6.

2. Façade-based data access

In this Section, we provide background information on façade-based data access as introduced in [15], including the specification of Façade-X and its mappings to CSV and JSON, that we use in our experiments.

2.1. Approach description

[15] proposed to provide access to heterogeneous sources in SPARQL [18] by relying on the notion of *façade* [19] as *"an object that serves as a front-facing interface masking more complex underlying or structural code"*[1]. Such a façade acts as a generic meta-model allowing (a) to inform the development of transformers from an open ended set of formats,

[1] See also The Facade Design Pattern: https://en.wikipedia.org/wiki/Facade_pattern (accessed 01/05/2022).

and (b) to generate RDF content in a consistent and homogeneous way. The meta-model that drives our approach is called Façade-X, while the implementation of the approach, supporting files in a variety of formats, is a system named SPARQL Anything. Façade-X is based on a set of basic data structures that are composed together: CONTAINMENT (inspired by the GoF pattern Composite), ORDERING (an unbound list), KEY-VALUES (a map), and TYPING (the unary predicate of description logic). As such, it can be expressed by using a subset of the RDF specification [14]: resources, types, properties, and container membership properties. The approach is implemented in SPARQL (with no syntax extensions), by overriding the SERVICE operator with a *virtual endpoint*, which serves data extracted from legacy formats, but structured according to Façade-X. Through this method, it is possible to provide access to non-RDF resources directly from SPARQL, by leveraging the SERVICE clause, which is extended through the new IRI scheme `x-sparql-anything`. Such IRI schema allows to include an open-ended set of key-value parameters, for example, pointing to the location of the resource, the expected media-type, and any other option provided by the Façade-X engineer. A SPARQL 1.1 query engine will execute the SERVICE clause by accessing the non-RDF resource applying the façade, evaluating the operations in the SERVICE clause, and returning the query solution, continuing the execution of the remaining steps in the query plan. The approach does not commit to a specific way of applying the façade. Intuitively, approaches span from materialising an RDF view on which to execute the query, to rewriting the query to fit a formalism applicable to the source format. In this paper, we focus on the first family of approaches.

2.2. Problem formalisation

A formal definition of façade-based data access as *materialised views* follows.

Let Q be the set of all possible queries (SERVICE clauses), G the set of all possible graphs, N the set of all possible graph names, R the set of all possible resources and DS the set of all data sources. Note that, we consider a *resource* anything accessible from a URL (including static files and dynamic responses generated by web APIs) and distinguish it from its content, that we name *data source*. We define:

(i) D as a collection of named graphs;

$$D \subseteq N \times G$$

(ii) F is a set of façade functions, where each $f \in F$ associates a data source from the resource ($ds \in DS$) and a query ($q \in Q$) with a graph $g \in G$, according to a façade;

$$F = \{f | f : DS \times Q \rightarrow G\}$$

(iii) A (i.e. the algorithm) is a function that given a resource ($r \in R$), a façade function ($f \in F$), and a query ($q \in Q$), returns a collection of named graphs including the graphs required to answer the query (i.e. one of the possible subsets of $N \times G$).

$$A : R \times F \times Q \rightarrow 2^{N \times G}$$

Additionally, given a query $q \in Q$, a resource $r \in R$ and its data sources $ds \in DS$ (note that $ds \subset r$ indicates that r includes ds), we define:

(i) $g^*_{ds,q} \in G$ as the graph which contains the minimal (optimal) set of triples required to answer q on ds;

(ii) $D^*_{r,q}$ as the collection of minimal named graphs required to answer q on r, in other words, $D^*_{r,q}$ is the RDF dataset containing all the quads required to answer q on r.

$$D^*_{r,q} = \{(n, g^*_{ds,q}) | ds \subset r \text{ and } n \in N \text{ and } g^*_{ds,q} \in G\}$$

It is worth noticing that given a query and a resource neither A nor any $f \in F$ has to return an optimal response (i.e. $D^*_{r,q}$ and $g^*_{ds,q}$), but they can return any super set of the optimum satisfying the query (i.e. any $g \in G$ such that $g^*_{ds,q} \subseteq g$). Moreover, the formalisation does not prescribe how to manage the result of the algorithm or how to evaluate the query on it, thus motivating the research on possible strategies for the concrete implementation of the approach. In this paper, we experiment with a single formalisation (i.e. Façade-X) and four algorithms (i.e. *in-memory+complete*, *in-memory+triple-filtering*, *sliced+triple-filtering* and *on-disk+triple-filtering*).

2.3. Façade-X specification

Source data is supposed to be translated into Façade-X. The Façade-X meta-model was designed by selecting a small set of primitive data structures: typing, key-value maps, and sequences. Façade-X defines two types of objects: containers and values. Containers can be typed, and one container in the dataset is always of type `root` (the only primitive specified). Values can

have any datatype. Containers include a set of unique slots, either labelled as strings or as integer numbers. A slot is filled by another container or by a value. An RDF specification of Façade-X uses two namespaces and associated preferred prefixes:

```
@prefix fx: <http://sparql.xyz/facade-x/ns/>
@prefix xyz: <http://sparql.xyz/facade-x/data/>
```

The first is used to express the primitive `fx:root`. String slots are RDF properties generated with the `xyz:` namespace, where the local name is supposed to be the string labelling the slot in the data source (for example, a JSON property)[2].

Instead, integer slots (sequences) are represented with instances of `rdf:ContainerMembershipProperty`: `rdf:_1`, `rdf:_2`, ... `rdf:_n`. Finally, values are `rdf:Literal`, while containers can be either `IRI`s or blank nodes (the specification does not enforce the use of of either, leaving both options to the Façade-X engineer, including the possibility of switch between the two with a tool option).

With these set of components, a Façade-X engineer is supposed to design connectors to an open-ended set of resource types, leaving to the knowledge engineer the freedom of accessing those data sources as-if they were RDF.

2.3.1. *Mappings to CSV and JSON*

A reference implementation of the approach is the system SPARQL Anything, which currently provides access to an extensive number of file formats.

In this paper, we focus on two popular formats, CSV and JSON. Here we provide an informal, intuitive description of the mappings, while referring to [8] for a formal specification.

Mappings to CSV. A comma-separated values (CSV) file is a text file that uses a comma to separate an ordered sequence of values in a data record and a carriage return to separate the data records. A CSV can be represented as a *list of lists* in which the outer list captures the sequence of data records (represented as containers), while the inner list captures the sequence of primitive values within a record.

As a example please consider the following CSV file located at https://sparql-anything.cc/examples/simple.csv.

[2]In the case data sources are URI-aware, for example in the case of XML namespaces, the translation engine may reuse the same URIs.

```
email,name,surname
laura@example.com,Laura,Grey
craig@example.com,Craig,Johnson
mary@example.com,Mary,Jenkins
jamie@example.com,Jamie,Smith
```

The resulting Façade-X RDF is the following.

```
@prefix fx: <http://sparql.xyz/facade-x/ns/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-
    syntax-ns#> .

[ a fx:root ;
  rdf:_1 [ rdf:_1 "laura@example.com" ;
           rdf:_2 "2070" ;
           rdf:_3 "Laura" ;
           rdf:_4 "Grey"
         ] ;
  rdf:_2 [ rdf:_1 "craig@example.com" ;
           rdf:_2 "4081" ;
           rdf:_3 "Craig" ;
           rdf:_4 "Johnson"
         ] ;
  rdf:_3 [ rdf:_1 "mary@example.com" ;
           rdf:_2 "9346" ;
           rdf:_3 "Mary" ;
           rdf:_4 "Jenkins"
         ] ;
  rdf:_4 [ rdf:_1 "jamie@example.com" ;
           rdf:_2 "5079" ;
           rdf:_3 "Jamie" ;
           rdf:_4 "Smith"
         ]
] .
```

However, it is common practice to use the first row of a CSV as a header. In that case, Façade-X can use named properties for the inner list rather then container membership properties. Therefore, the CSV file is interpreted as following.

```
@prefix fx: <http://sparql.xyz/facade-x/ns/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-
    syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>
    .
@prefix xyz: <http://sparql.xyz/facade-x/data/>
    .

[ rdf:type fx:root ;
  rdf:_1  [ xyz:email "laura@example.com" ;
            xyz:name  "Laura" ;
            xyz:surname "Grey"
          ] ;
  rdf:_2  [ xyz:email "craig@example.com" ;
            xyz:name  "Craig" ;
            xyz:surname "Johnson"
          ] ;
  rdf:_3  [ xyz:email "mary@example.com" ;
            xyz:name  "Mary" ;
            xyz:surname "Jenkins"
          ] ;
  rdf:_4  [ xyz:email "jamie@example.com" ;
            xyz:name  "Jamie" ;
            xyz:surname "Smith"
          ]
] .
```

Mappings to JSON. JSON is a text format which is built on two structures: collections of key/value pairs (a JSON object) and ordered lists (a JSON array). Both JSON objects and JSON arrays are representable as containers. While objects will be translated into containers having named properties, arrays will use container membership properties. Finally, JSON data types map to common XSD datatypes.

As a example please consider the following JSON file located at https://sparql-anything.cc/examples/tvseries_simple.json.

```
[
  {
    "name":"Friends",
    "stars":[ "Jennifer Aniston", "Courteney Cox"
         , "Lisa Kudrow", "Matt LeBlanc", "
         Matthew Perry", "David Schwimmer"]
  },
  {
    "name":"Cougar Town",
    "stars":[ "Courteney Cox", "David Arquette",
         "Bill Lawrence", "Linda Videtti
         Figueiredo", "Blake McCormick" ]
  }
]
```

The resulting Façade-X RDF is the following.

```
@prefix fx: <http://sparql.xyz/facade-x/ns/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-
     syntax-ns#> .
@prefix xyz: <http://sparql.xyz/facade-x/data/>
     .

[ rdf:type fx:root ;
  rdf:_1  [ xyz:name "Friends" ;
            xyz:stars [ rdf:_1 "Jennifer Aniston"
                 ;
                      rdf:_2 "Courteney Cox" ;
                      rdf:_3 "Lisa Kudrow" ;
                      rdf:_4 "Matt LeBlanc" ;
                      rdf:_5 "Matthew Perry" ;
                      rdf:_6 "David Schwimmer"
                    ]
          ] ;
  rdf:_2  [ xyz:name "Cougar Town" ;
            xyz:stars [ rdf:_1 "Courteney Cox" ;
                      rdf:_2 "David Arquette" ;
                      rdf:_3 "Bill Lawrence" ;
                      rdf:_4 "Linda Videtti
                           Figueiredo" ;
                      rdf:_5 "Blake McCormick"
                    ]
          ]
] .
```

3. Strategies for executing Façade-X queries

In this Section, we refer to the formalisation of façade-based data access provided in Section 2 and elaborate on possible execution strategies. We consider an example scenario, taken from the GTFS benchmark used later in our experiments. The data represents geospatial information of the transport network of a city. In this scenario, RML mappings originally designed in the benchmark are translated into Façade-X `SERVICE` clauses and embedded in the SPARQL queries. The following lists how such data could be provided in two popular open data formats: CSV and JSON.

```
shape_id,shape_pt_lat,shape_pt_lon,
     shape_pt_sequence,shape_dist_traveled
00o,39248,39208,1008430,8429
00k,8469,8429,1010130,21148
00c,39798,39758,1001667,17663
...
```

```
[{ "shape_id": "00o",
   "shape_pt_lat": "39248",
   "shape_pt_lon": "39208",
   "shape_pt_sequence": "1008430",
   "shape_dist_traveled": "8429" },
   ... ]
```

The query in Figure 1, also elaborated from the benchmark, selects the data from the source and after generating IRIs for entities and fixing data types, returns the results[3].

Figure 2 illustrates the SPARQL query execution process. In the diagram, we use the convention of representing processes performed by components that are *given*, the blocks with grey background and plain line (for example, the SPARQL query engine or the format-specific parser library), and processes that are designed by the proposed approaches, in green with a dashed border and a bold text.

The system receives the input query and analyses it producing a query plan, combining the various operations, considering their dependencies. The example query is interpreted as in the following algebra:

```
(project (?shape ?shapePoint ?shape_pt_lat ?
     shape_pt_lon ?shape_pt_sequence)
  (extend
    ( (?shape (fx:entity shape: ?shape_id))
     (?shapePoint (fx:entity point: ?shape_id "-
          " ?shape_pt_sequence))
     (?shape_pt_lat (xsd:double ?shape_pt_lat_in
          ))
     (?shape_pt_lon (xsd:double ?shape_pt_lon_in
          )))
    (service <x-sparql-anything:...>
       (bgp
         (triple ??0 xyz:shape_id ?shape_id)
         (triple ??0 xyz:shape_pt_sequence ?
              shape_pt_sequence)
```

[3]The function `fx:entity` in the figure is a shorthand for `IRI(CONCAT(STR(),STR(),...)))`

```
PREFIX fx: <http://sparql.xyz/facade-x/ns/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX xyz: <http://sparql.xyz/facade-x/data/>
PREFIX shape: <http://transport.linkeddata.es/madrid/metro/shape/>
PREFIX point: <http://transport.linkeddata.es/madrid/metro/shape_point/>

SELECT ?shape ?shapePoint ?shape_pt_lat ?shape_pt_lon ?shape_pt_sequence
WHERE {
  SERVICE <x-sparql-anything:location=./shapes.csv,csv.headers=true> {
  # or <x-sparql-anything:location=location=shapes.json>
       [] xyz:shape_id ?shape_id ; xyz:shape_pt_sequence ?shape_pt_sequence ; xyz:shape_pt_lat ?
            shape_pt_lat_in ; xyz:shape_pt_lon ?shape_pt_lon_in .
  }
  BIND ( fx:entity( shape:, ?shape_id ) AS ?shape ) .
  BIND ( fx:entity( point:, ?shape_id ,"-", ?shape_pt_sequence ) AS ?shapePoint )
  BIND ( xsd:double(?shape_pt_lat_in) AS ?shape_pt_lat ) .
  BIND ( xsd:double(?shape_pt_lon_in) AS ?shape_pt_lon ) .
}
```

Fig. 1. SPARQL query with façade-based data access.

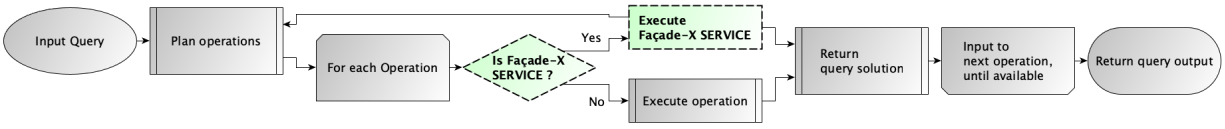

Fig. 2. Query execution flowchart

```
11          (triple ??0 xyz:shape_pt_lat ?
                shape_pt_lat_in)
12          (triple ??0 xyz:shape_pt_lon ?
                shape_pt_lon_in)))))
```

The algebra illustrates the dependencies between the various operations, where outer operations depend on the nested ones. Sibling operations are not ordered, however, operations evaluating variables are performed before operations relying on those variables. In the example, the `SERVICE` operation will be executed before the assignments in the `EXTEND` operation (e.g. to evaluate `?shape_id` needed to execute the function and assign `?shape`). When an operation is executed, the resulting bindings are streamed as input to the next one, until all operations are completed. All the operations are performed in compliance with the SPARQL 1.1 specification, except the `SERVICE <x-sparql-anything:...>` clause, focus of this paper.

In the definition provided by [15] and reported in Section 2, *A* is the algorithm that matches a resource *R* with a Façade function *F* to return an RDF dataset that can resolve the query *Q* (the operation `BGP` in the example). *R* is defined as a collection of *data sources DS*, each one to be transformed into a *G* by the function *F*. In our examples, *R* are the JSON or CSV files including one *DS* each[4].

[4]Other types of resources may include more that one *DS*. It is the case of file archives including multiple files or Excel spreadsheets having more sheets

We focus on two approaches for implementing *A* by producing *materialised views*, on which the `SERVICE` clause is executed. The first performs a complete transformation of each *DS* before executing the query. The second, segments the *DS* and applies the query on each one of the segments, in sequence, passing the query the solution as a single stream to the next SPARQL operation in the algebra.

3.1. Complete materialised view

The first method transforms the data before executing the query, only one time, on the *complete materialised view*. The second method, that we call *slicing*, partitions the data source, generates one materialised view for each partition, and executes the query on each one of the views, appending each iteration to the returning query solution.

In addition, we consider two ways of implementing *F*. The first translates the whole data source into RDF according to Façade-X . The second, that we name *triple-filtering*, inspects the `SERVICE` clause and only generates quads that match any of the patterns of any `BGP` operation included. It is worth noting how this method will always generate a set of triples which in-

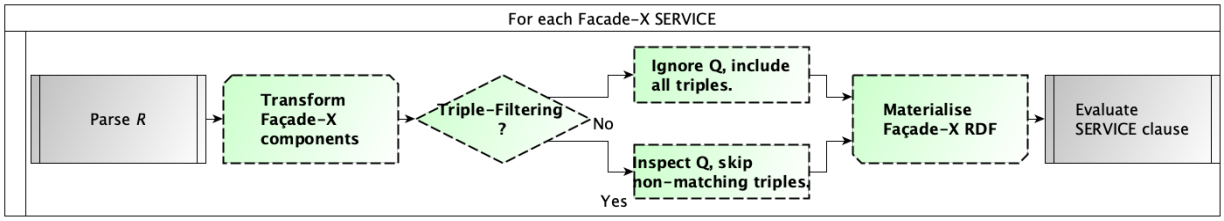

Fig. 3. Complete materialised view strategy. The input data sources are fully transformed into RDF according to the façade, before executing the query in the `SERVICE` clause. The Materialize Façade-X RDF operation can be equally performed *in-memory* or *on-disk*.

cludes all triples needed to answer the query, without necessarily be the optimal set (only the triples needed).

Figure 3 illustrates the *complete materialised view* strategy for evaluating Façade-X service clauses. In practice, a complete materialised view can be generated both *in-memory* and *on-disk*. A format-specific parser accesses the resource *R* and streams the data to a transformer, implementing Façade-X mappings. Façade-X components are translated into triples and loaded into an in-memory RDF dataset. If *triple-filtering* is applied, the `SERVICE` clause is inspected and potential triples matched against any quad patterns expressed. In the case of our exemplary query (see Figure 1), the data related to the CSV column (and equivalent JSON property) `shape_dist_traveled` will not be materialised. Finally, the operations withing the `SERVICE` clause – the `BGP` in the example – are evaluated on the in-memory view. The query solution is streamed back to the query executor, and the execution continues.

3.2. Sliced materialised view

The complete materialised view can be resource-intensive. In addition, SPARQL queries may include multiple façade-based data access operations, accumulating many complete views, either *in-memory* or *on-disk*, before the operations are actually evaluated (and query solution streamed) to the client application. Indeed, the application scenarios are limited to data sources that translate to RDF datasets fitting the amount of memory available. For these reasons, we design an alternative method to implement façade-based data access (the function *A*). Specifically, we consider how resources can be interpreted as collections of data sources by applying a segmentation method. Such a method can be derived automatically or can be given by the user. For example, a CSV file can be automatically interpreted as a collection of rows (Façade-X containers). JSON files can have a JSON Array as top element (the "root" container, in Façade-X terminology), allowing an automatic slicing approach. Alternatively, users may provide some input parameter, for example, a JsonPath expression to slice the data source or indicate a number of rows to include in each CSV slice.

In Figure 4 we describe a *sliced materialised view* strategy. The *slice* operation is applied to the resource *R* and a materialised view generated for each one of the partitions by following the same process as in the complete materialised view method. Similarly, the amount of generated triples can be additionally reduced by applying *triple-filtering*. The query is executed on each one of the slices as multiple, distinct *DS* , and the query solutions are appended to the returning stream.

3.3. Discussion

We can already make some considerations on the differences between these methods. A complete materialised view approach may fail on large data sources for insufficient memory or result in excessive I/O operations and disk space. Triple-filtering should be beneficial in reducing the resources requirements. Such benefit will be proportionate to the amount of triples that are not loaded in memory, and vary across queries. However, in the first case (complete), both the transformation and query evaluation operations are performed once and for all. In the second case (sliced), those two operations are performed for each slice (for example, each one of the CSV rows). Therefore, we can expect that a sliced materialised view strategy is less *time-efficient* than a complete materialised view, since all the computational effort for preparing an in-memory view (independently from its size) and for evaluating the query (independently from the matching data) will be repeated many times. In contrast, memory footprint should favour a sliced approach, since the memory required for materialising a slice of data will necessarily be much smaller than the one required to materialise all data, and such memory can be freed-up after

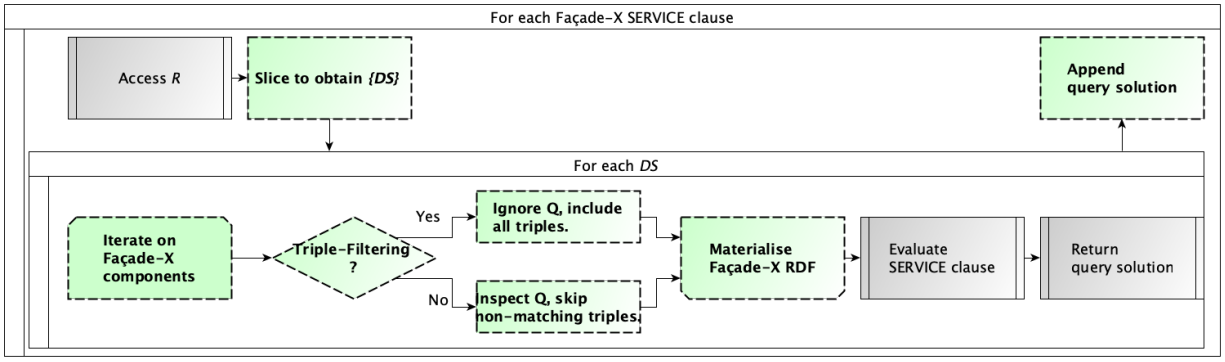

Fig. 4. Sliced materialised view

each one of the query evaluation on the slice. In what follows, we pragmatically evaluate these hypotheses. However, we don't consider the option of generating slices on-disk, as the further multiplication of I/O operations would make that inefficient by design.

4. Evaluation

We conducted a comparative analysis of the implementation of: (i) the two execution strategies, i.e. *complete materialised view* and *sliced*; (ii) the *triple filtering* optimisation option; (iii) and, the in-memory and on-disk execution. To evaluate our hypotheses on the time and memory footprint of the approaches, we designed a benchmark which executes a set of queries of varying complexity under time and memory constraints. Specifically, we tested 4 experimental regimes and 56 different experimental settings (discussed in Section 4.3). The queries were executed under the following regimes:

1. In-memory execution over a complete materialised view (in-memory+complete);
2. In-memory execution optimised by a triple-filtering approach (in-memory+triple-filtering);
3. In-memory execution over a sliced materialised view and optimised by triple-filtering (sliced+triple-filtering);
4. On-disk execution optimised by triple-filtering (on-disk+triple-filtering).

It is worth noticing that not all the combinations of the three options were experimented since: *(i)* triple-filtering is a straightforward optimisation of the execution which it is reasonable to keep enabled except under the greedy regime (i.e., in-memory execution over a complete materialised view); *(ii)* sliced materialised view is inconvenient by design under on-disk regimes, since it requires a on-disk dataset to be created for each slice.

During the experiments, time is constrained by setting a 5 minutes timeout for each query execution and the memory is limited by controlling the heap size given to the Façade-X engine. To push the engine to the limit, we also executed the queries on inputs of increasing size. The behaviour of the proposed approaches is monitored by recording, for each query, the total execution time, and the CPU usage and the resident memory used by the Façade-X engine.

4.1. Benchmark preparation

The analysis relied on the GTFS-Madrid benchmark [10], a virtual knowledge graph access benchmark aimed at assessing the performance of OBDA engines. The benchmark is composed of the following elements: *(i)* A collection of data sources in different formats (i.e. CSV, JSON, XML, SQL); *(ii)* A set of RML [16] mappings that describe how to transform input sources into an RDF graph compliant with the Linked GTFS ontology[5]; *(iii)* A set of 18 `SPARQL SELECT` queries of varying complexity which interrogates transformed RDF data; *(iv)* A set of metrics. Moreover, GTFS provides a data generator to scale up the original data in terms of size, and distribute the datasets over different formats.

We extend the GTFS-Madrid-bench to evaluate the façade-based data access engines as follows. The extension consists of: *(i)* a set of query templates that translate the GTFS' queries into Façade-X queries; *(ii)* a query executor which fires the queries and measures the performance of the Façade-X based engine; *(iii)* a series of python scripts aimed at analysing the

[5] https://github.com/OpenTransport/linked-gtfs

measurements and producing summary charts and tables (also available as supplemental material).

4.2. Façade-X queries and experiments templates.

We implemented the 18 SPARQL queries of the benchmark as Façade-X queries, leaving a set of open configuration parameters, to be set on the different experimental regimes. Each query template aims at *(i)* instructing the Façade-X engine to mimic the transformation of the input sources into RDF according to the GTFS' declarative mappings; and, *(ii)* returning the bindings of the variables as defined in the GTFS' queries. The query templates are then turned into effective Façade-X queries by a dedicated query generator script. Specifically, for each pair ⟨format, size⟩ given as an argument to the GTFS' data generator, the script generates four Façade-X queries, one for each experimental regime, namely: in-memory+complete, in-memory+triple-filtering, sliced+triple-filtering, on-disk+triple-filtering. An example of such a query is shown in the Figure 1. As for the in-memory experimental regimes, the related query template includes the following SERVICE IRI:

```
<x-sparql-anything:location=../result/datasets
    /\%format/\%size/SHAPES.\%format,\%param,
    slice=\%slice,strategy=\%strategy>
```

where *strategy* refers to enabling the triple-filtering or not. As for the on-disk regime, the IRI also specifies the path where the triples are stored during the query execution via the `ondisk` option.

The `SELECT` clause specifies all the variables returned in the corresponding GTFS query. The `WHERE` clause contains a `SERVICE` clause for each source file to be queried (one in the example). The `SERVICE` IRI template is specified according to the Façade-X IRI schema and contains variables of the form `%variable_name` which are substituted by the query generator script. Moreover, the `WHERE` clause contains:

- *(i)* a BGP, compliant with the Façade-X (meta)model [15], which selects from the input files the fields needed for answering the query;
- *(ii)* a list of `BIND` instructions which build the IRIs to be returned, consistently with the benchmark RML mappings;
- *(iii)* possibly one or multiple `FILTER`, `UNION`, `GROUP BY` or `ORDER BY` clauses to further refine the results (as defined by the GTFS queries).

Benchmark queries. Table 1 reports the number of records (i.e. CSV records or JSON objects) of each data source at size 1 and the characteristics of the queries in terms of number and kinds of sources, number of triple patterns of the query, number of `FILTER` clauses, presence of `UNION`, `GROUP BY` and `ORDER BY`. Queries on the "SHAPES" source (i.e. q1 and q9) and queries on multiple sources (such as q7, q8, and q12) are challenging since they require the Façade-X engine to go through many data files and/or records to evaluate the query. Notably, queries q2 and q3 insist on half of the fields of the source file (q2 and q3 query 5/6 of the 12 fields of the STOPS source). Therefore, the execution of these queries is an ideal benchmark for the assessment of the triple filtering mechanism. We expect that other queries, such as q4, q5, and q11, to be easily executed since they involve sources having fewer records.

Correctness of the executions. We ensured that all the queries were designed correctly. First, we generated the GTFS sources for the JSON and CSV formats and size 1. Then, we transformed the sources into RDF using the GTFS mapping and the RMLMapper and we uploaded the results to an instance of the Fuseki SPARQL endpoint. Finally, we compared the results given by our engines with the queries returned by Fuseki. All the queries returned the expected results.

Query Executor. The query executor submits the queries and measures the performance of the Façade-X based engine. Specifically, it iterates over the Façade-X queries, it submits each query three times for each experimental heap size.Then, it measures the execution time, the average usage of the CPU, and the resident memory used by the Façade-X engine. To assess the behaviour under memory constraints, we tested the queries with a decreasing heap sizes. Furthermore, we set a 5 minutes timeout for each query. For practical reasons, if a query exceeds the timeout or the memory limit the executor avoids submitting further the query with less memory.

The query executor is implemented as a BASH script which executes SPARQL Anything through its command line interface. To avoid dependencies among different query runs (e.g. caching results), each query run was executed by a different command and the temporary directory storing the TDB used in the on-disk executions is wiped after each run. The query executor relies on Unix commands (ps and gdate) for measuring execution time, memory and cpu usage.

	Records	Fields	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14	q15	q16	q17	q18
AGENCY	1	7				✓														
CALENDAR	5	10								✓			✓							✓
CALENDAR_DATES	70	3					✓			✓			✓					✓		
FEED_INFO	1	6																		
FREQUENCIES	855	5																	✓	
ROUTES	13	9				✓		✓	✓	✓				✓					✓	✓
SHAPES	58K	5	✓								✓									
STOPS	1.2K	12		✓	✓				✓	✓				✓	✓	✓	✓			
STOP_TIMES	2.3K	9							✓	✓		✓		✓		✓				
TRIPS	130	9							✓	✓	✓	✓	✓	✓		✓		✓	✓	✓
# of sources			1	1	1	2	1	1	4	6	2	2	3	4	1	3	1	2	3	3
# of triple patterns			4	5	6	9	3	2	14	14	8	3	8	9	5	7	2	6	8	9
# FILTER			0	1	1	0	2	1	1	0	1	1	3	0	0	0	1	2	0	0
UNION										✓										✓
GROUP BY														✓						
ORDER BY																	✓	✓		

Table 1

Number of records and number of fields per record of each data source at size 1 and features of the queries.

All the resources (queries, measurements etc.) relevant to the evaluation and useful for reproducing the experiments are available on the GitHub repository[6].

4.3. Experiments settings

The queries were executed under four experimental regimes (i.e. in-memory+complete, in-memory+triple-filtering, sliced+triple-filtering, on-disk+triple-filtering) and 56 experimental settings whose dimensions summarised below. We experimented with two types of data sources, (i.e. CSV and JSON), four GTFS sizes (i.e. 1, 10, 100, 1000), seven heap sizes (i.e. 256mb, 512mb, 1gb, 4gb, 8gb, 16gb and 32gb). The tests were executed on a MacBook Pro 2020 (CPU: i7 2.3 GHz, RAM: 32GB). We implemented the proposed approaches in Java extending the Apache Jena SPARQL processor (ARQ) [3], and rely on Apache Commons CSV [2] and Jackson [5] as CSV and JSON parsers, respectively. Source code of the query executor is on the GitHub repository[6]. The reference implementation of the experimented approaches can be downloaded from here the release page of the GitHub repository for the SPARQL Anything project[7].

4.4. Results

We report on the results of the experiments with the 4 experimental regimes and the 56 experimental settings. Tables 2, 3, 4, and 5 report the result with the different experimental regimes. Each table reports the number of queries successfully executed (green), those exceeding the timeout (orange) or the memory limit (red), and those not executed (black) (e.g. 15-0-1-2 - in Table 2 - row 256 and column 10 - means that 15 queries were successfully executed, 0 exceeded the timeout, 1 exceeded the memory limit and 2 were not executed because precedent executions raised an out of memory error with heap size 1024).

Figures 5 6 7 8 report on the time needed for evaluating each query over the input sources of size 1, 10, 100 and 1000 and under the four experimental regimes.Note that the Figures report only the time of the successful executions. Specifically, the height of the bar indicates the average execution time over the four memory limits and the error bar indicates the standard deviation among the measurements. Additionally, the supplemental material reports on the average usage of the CPU, and the resident memory used by the engine.

4.5. Discussion

We discuss the results presented in the previous section. Specifically, we evaluate façade-based data access on each *use case*: a benchmark query to a given format, with a memory requirement, on a data source of a given size.

Complete materialised view strategy hits memory limits. We observe a significant number of Out-Of-Memory Errors (OOMEs) for the experimental regimes with complete in-memory materialisation, i.e. 3.6% of the use cases raised an OOME (96.4% of use cases are supported).

[6] https://github.com/SPARQL-Anything/experiments/tree/main/gtfs

[7] https://github.com/SPARQL-Anything/sparql.anything/releases

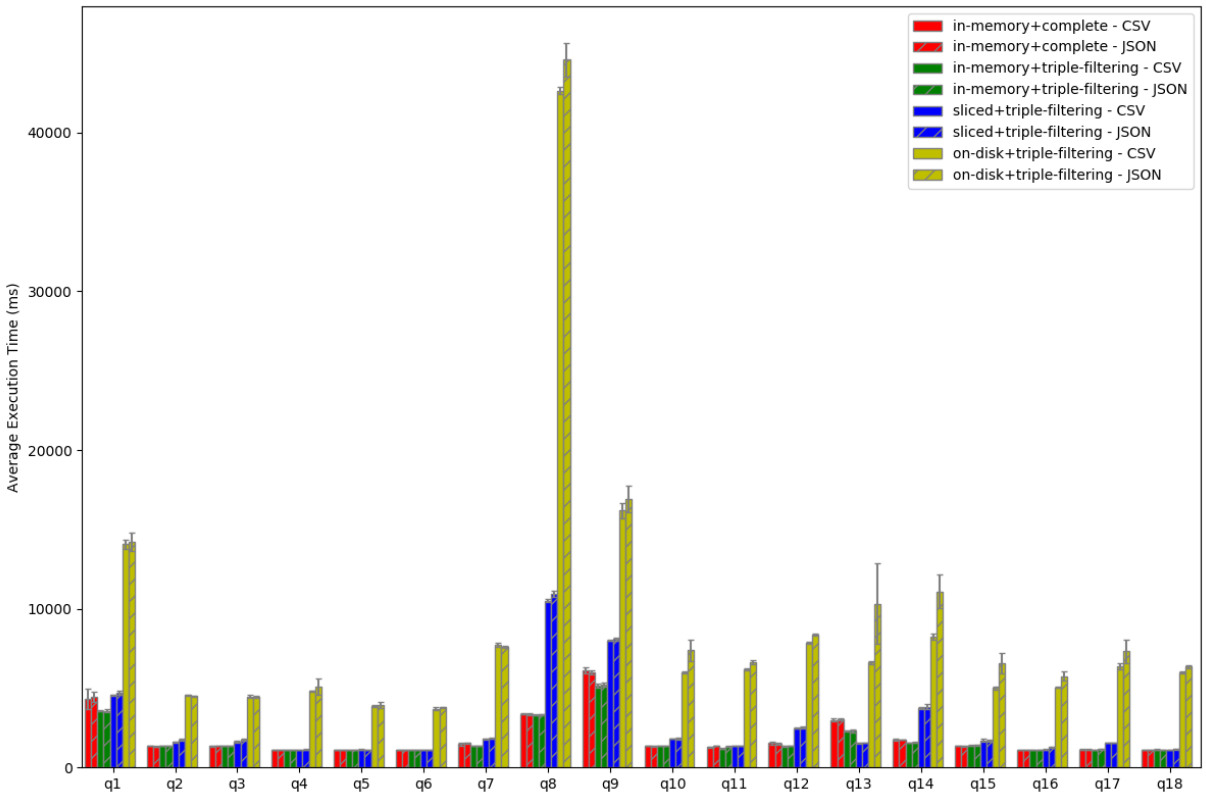

Fig. 5. Average execution time of queries over the sources of size 1 under the four experimental regimes.

	CSV				JSON			
	1	10	100	1000	1	10	100	1000
256	18-0-0-0	15-0-1-2	6-0-1-11	3-0-0-15	18-0-0-0	15-0-1-2	6-0-1-11	3-0-0-15
512	18-0-0-0	16-0-0-2	7-0-3-8	3-0-3-12	18-0-0-0	16-0-0-2	7-0-3-8	3-0-3-12
1024	18-0-0-0	16-0-2-0	10-0-4-4	6-0-1-11	18-0-0-0	16-0-2-0	10-0-4-4	6-0-1-11
4096	18-0-0-0	18-0-0-0	14-0-0-4	7-0-2-9	18-0-0-0	18-0-0-0	14-0-0-4	7-0-3-8
8192	18-0-0-0	18-0-0-0	14-0-1-3	9-0-0-9	18-0-0-0	18-0-0-0	14-0-1-3	10-0-0-8
16384	18-0-0-0	18-0-0-0	15-0-0-3	9-0-0-9	18-0-0-0	18-0-0-0	15-0-0-3	10-0-0-8
32768	18-0-0-0	18-0-0-0	15-3-0-0	9-9-0-0	18-0-0-0	18-0-0-0	15-3-0-0	10-8-0-0

Table 2

Results of the execution of the queries under the regime in-memory execution over a complete materialised view. Each cell reports the number of queries successfully executed (green), those exceeding the timeout (orange) or the memory limit (red), and those not executed (black).

Slicing and on-disk is effective for memory demanding use cases. Instead, applying slicing or storing the graph on-disk, all experiments proceed with the memory available (no observed OOMEs with slicing and a few - 0.3% - were observed with on-disk UCs). In addition, 69.8% of the use cases not supported by the complete approach were successfully complemented within the time limit of the experimental setting, bringing the total number of supported cases to 91%. The remaining 9% of use cases would require more than 5 minutes to complete. The slicing strategy enables façade-based data access to evaluate queries that couldn't be evaluated with a complete materialised view strategy. However, slicing and on-disk are time consuming strategies which require further investigation to explore possible optimisation of the implementation.

Triple filtering reduces the execution time. By con-

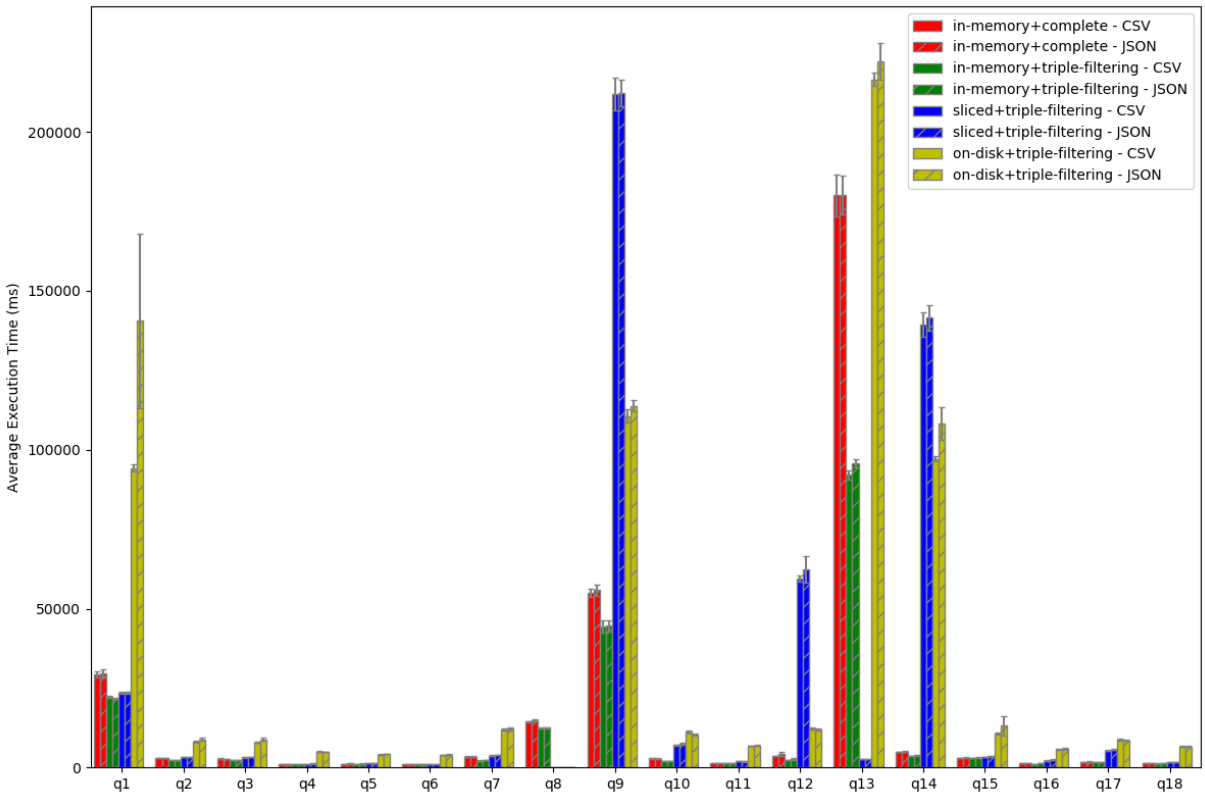

Fig. 6. Average execution time of queries over the sources of size 10 under the four experimental regimes.

	CSV				JSON			
	1	10	100	1000	1	10	100	1000
256	18-0-0-0	16-0-0-2	7-0-4-7	4-0-2-12	18-0-0-0	16-0-0-2	7-0-4-7	4-0-2-12
512	18-0-0-0	16-0-1-1	11-0-3-4	6-0-0-12	18-0-0-0	16-0-1-1	11-0-3-4	6-0-0-12
1024	18-0-0-0	17-0-1-0	14-0-1-3	6-0-4-8	18-0-0-0	17-0-1-0	14-0-1-3	6-0-4-8
4096	18-0-0-0	18-0-0-0	15-0-0-3	10-1-2-5	18-0-0-0	18-0-0-0	15-0-0-3	10-1-2-5
8192	18-0-0-0	18-0-0-0	15-0-0-3	13-1-0-4	18-0-0-0	18-0-0-0	15-0-0-3	13-1-0-4
16384	18-0-0-0	18-0-0-0	15-0-0-3	14-0-0-4	18-0-0-0	18-0-0-0	15-0-0-3	14-0-0-4
32768	18-0-0-0	18-0-0-0	15-3-0-0	12-6-0-0	18-0-0-0	18-0-0-0	15-3-0-0	14-4-0-0

Table 3

Results of the execution of the queries under the regime in-memory execution optimised by a triple-filtering approach. Each cell reports the number of queries successfully executed (green), those exceeding the timeout (orange) or the memory limit (red), and those not executed (black).

sidering the regimes without slicing and in-memory, the experiments with triple filtering lasted 67,223ms on average, while the runs without triple filtering took on average 58,208ms (15% more). However, the benefit of triple-filtering is query-dependent. The best results were obtained with `q2`, `q3`, `q7`, `q10` and `q12` in which the execution with triple filtering required less than half of the time (cf. supplemental material). These queries benefit considerably from triple filtering as they involve a limited number of source fields (q2 involves 5 fields among the 12 available, q3 involves 6 fields among the 12 available, q10 involves 3 among the 18 available, q12 involves 9 among the 39 available). Moreover, the queries involve some of largest input sources (STOP and STOP_TIMES) and, as a result, the benefits of the triple filtering are more visible.

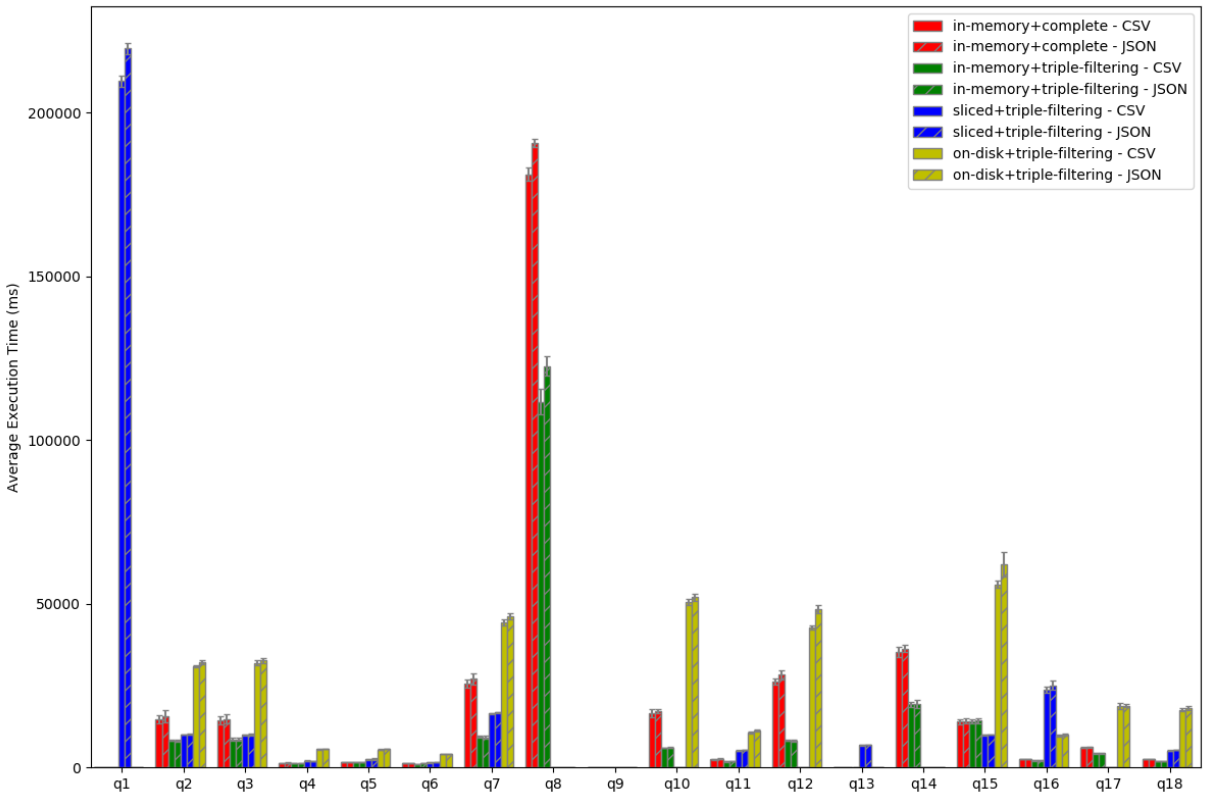

Fig. 7. Average execution time of queries over the sources of size 100 under the four experimental regimes.

	CSV				JSON			
	1	10	100	1000	1	10	100	1000
256	18-0-0-0	17-0-0-1	12-0-0-6	10-0-0-8	18-0-0-0	17-0-0-1	12-0-0-6	10-0-0-8
512	18-0-0-0	17-0-0-1	12-0-0-6	10-0-0-8	18-0-0-0	17-0-0-1	12-0-0-6	10-0-0-8
1024	18-0-0-0	17-0-0-1	12-0-0-6	10-0-0-8	18-0-0-0	17-0-0-1	12-0-0-6	10-0-0-8
4096	18-0-0-0	17-0-0-1	12-0-0-6	10-0-0-8	18-0-0-0	17-0-0-1	12-0-0-6	10-0-0-8
8192	18-0-0-0	17-0-0-1	12-0-0-6	10-0-0-8	18-0-0-0	17-0-0-1	12-0-0-6	10-0-0-8
16384	18-0-0-0	17-0-0-1	12-0-0-6	10-0-0-8	18-0-0-0	17-0-0-1	12-0-0-6	10-0-0-8
32768	18-0-0-0	17-1-0-0	12-6-0-0	10-8-0-0	18-0-0-0	17-1-0-0	12-6-0-0	10-8-0-0

Table 4

Results of the execution of the queries under the regime in-memory execution over a sliced materialised view and optimised by triple-filtering. Each cell reports the number of queries successfully executed (green), those exceeding the timeout (orange) or the memory limit (red), and those not executed (black).

*Complete materialised view strategy is time efficient.*The complete materialised view strategy is time-efficient, when memory is available. For example, the average execution time of successfully executed queries under the complete in-memory regimes (without slicing) is 37,952ms and increases to 44,258ms with slicing or on-disk. In light of these observations, we further investigated this issue and noticed that the complete materialised strategy better exploited the CPU, that is, %202[8] versus %50 on average during the experiments with input size 1000 (the experimental setting that stresses the CPU the most).

[8]The percentange refers to the CPU utilization of the process as reported by the ps command. 202% means that the process is fully occupying two CPUs plus a little of another.

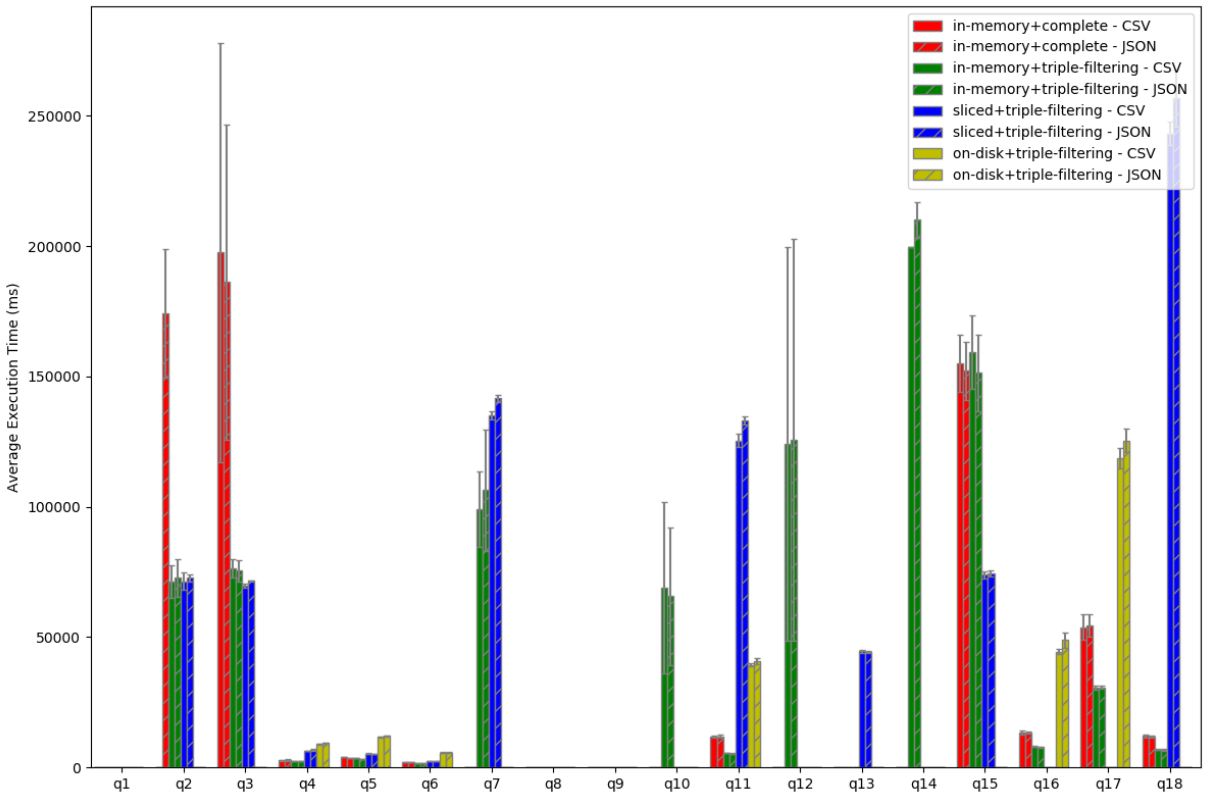

Fig. 8. Average execution time of queries over the sources of size 1000 under the four experimental regimes.

	CSV				JSON			
	1	10	100	1000	1	10	100	1000
256	18-0-0-0	17-0-0-1	11-0-2-5	6-0-0-12	18-0-0-0	17-0-0-1	11-0-2-5	6-0-0-12
512	18-0-0-0	17-0-0-1	13-0-0-5	6-0-0-12	18-0-0-0	17-0-0-1	13-0-0-5	6-0-0-12
1024	18-0-0-0	17-0-0-1	13-0-0-5	6-0-0-12	18-0-0-0	17-0-0-1	13-0-0-5	6-0-0-12
4096	18-0-0-0	17-0-0-1	13-0-0-5	6-0-0-12	18-0-0-0	17-0-0-1	13-0-0-5	6-0-0-12
8192	18-0-0-0	17-0-0-1	13-0-0-5	6-0-0-12	18-0-0-0	17-0-0-1	13-0-0-5	6-0-0-12
16384	18-0-0-0	17-0-0-1	13-0-0-5	6-0-0-12	18-0-0-0	17-0-0-1	13-0-0-5	6-0-0-12
32768	18-0-0-0	17-1-0-0	13-5-0-0	6-12-0-0	18-0-0-0	17-1-0-0	13-5-0-0	6-12-0-0

Table 5

Results of the execution of the queries under the regime on-disk execution optimised by triple-filtering. Each cell reports the number of queries successfully executed (green), those exceeding the timeout (orange) or the memory limit (red), and those not executed (black).

Increasing memory does not generally improve time performance. Table 6 reports on the average execution time of the experimental configurations (i.e. experimental regimes and experimental settings) successfully completed with all heap sizes (e.g. the query q18, over JSON data sources of size 1000, under the in-memory regime over a sliced materialised view optimised with triple filtering was successfully executed with all heap sizes).

We observe that increasing the heap size doesn't affect the execution time significantly in the reference benchmark queries. On the contrary, the queries executed with the largest heap size took slightly longer on average (since the operating system needs more time to reserve the necessary memory).

Results are format independent. It is worth noticing that the average execution time of the successfully executed queries is comparable for both experiment-

256MB	512MB	1GB	4GB	8GB	16GB	32GB
17,560ms	17,323ms	17,325ms	17,463ms	17,388ms	17,375ms	17,770ms

Table 6
Average execution time of the experimental configuration successfully completed with all heap sizes.

ing formats, i.e. 19,469ms for CSV and 20,297ms for JSON. We have also observed a similar behaviour in terms of number of timeouts (2.7% of the experiments for CSV and 2.6% for JSON) and OOMEs (1.8% of the experiments for CSV and 1.9% for JSON). In the light of this observations, we can speculate that the behaviour observed in these experiments generalises to other formats.

To summarise, the findings suggest the following guidelines for choosing the startegy of execution for façade-based data access:

- *Complete materialised view strategy* is faster than the slicing approach, but it can be problematic on large inputs.
- *Triple filtering* reduces the memory usage and the execution time, therefore, it can be adopted as a default optimisation strategy.
- *Slicing* and using an *on-disk* graph for evaluating the query are effective solutions for coping with limited memory requirements and large data sources.
- Low-level components (such as the parsing components, the query engine and other components of the façade engine) have a stable impact on the performance on the different regimes. Our results are format-independent.

5. Related work

Façade-based data access was originally introduced by [15]. The principal benefit of this approach lies in the fact that knowledge engineers familiar with SPARQL, which are not necessarily software developers [25], can apply their expertise to integrate external data without the need of knowing a format-specific formalism (such as JsonPath) or learn new languages. We refer to [15] for a discussion on how façade-based data access compares to alternative approaches from a user point of view. In [15], authors compare a prototype implementation, akin to our *complete materialised view*, with two alternative tools (RML Mapper and SPARQL Generate), demonstrating how façade-based data access is generally sustainable. However, the present work is the first study dedicated to execution strategies for implementing façade-based data access.

Here, we consider related work on methods for re-engineering data files into RDF, in relation to the execution strategies presented in this paper. We do not scrutinise all systems for knowledge graph construction, and refer the reader to [7] for a broad overview.

Several tools are available for automatically transforming data sources of various formats into RDF (Any23 [1], JSON2RDF [6], CSV2RDF [4] to name a few). While these tools have a similar goal (i.e. enabling the user to access a data source as if it was in RDF), they do not allow direct querying from SPARQL.

Approaches use an *intermediate language* to develop mappings to non-RDF formats. In [23], the process focuses on re-engineering the meta-model, described as an ontology and a set of transformation rules, for obtaining an RDF dataset which will be in turn refactored to the end product, domain-oriented dataset. However, this approach requires to define a well-specified ontology for each of the formats and produce a different RDF structure for each of the meta-models involved. R2RML mappings can be interpreted in different ways, for example, supporting query-rewriting strategies to relational databases [9][9]. RML [16] is proposed as a general mapping language, whose features support some popular formats. Engines such as the RML Mapper evaluates mappings via a two-step execution flow. First, an iterator expression (for example a JsonPath) selects portions of the data source. Second, mappings are evaluated for each one of the matches and triples streamed back to the client application. This execution flow has the advantage of being incremental, allowing the streaming of data points from the source to the output. However, this opens additional challenges, for example, in terms of generation of duplicates and limited pre-processing capabilities, tackled by systems such as MapSDI [20] and MorphCSV [11]

The SheX-ML [17] language is structured via an *iterator* operator (matching portions of the data source), *expressions* to manipulate matching data, and *shapes*

[9]However, façade-based data access on relational databases is not the scope of this paper, hence we don't discuss further this topic.

to generate RDF data. SPARQL Generate [21] proposes to enable direct access to non-RDF data by extending the SPARQL syntax with a set of new operators: GENERATE, SOURCE, and ITERATOR. The captured data fragment populates a custom SPARQL node type. Custom functions perform ad-hoc operations on the supported formats, for example, to access the cells in a CSV row.

In this article we aim at exploring materialisation approaches to façade-based data access in terms of performance and memory requirements, and not to improve the efficiency of knowledge graph construction pipelines in general. Although with a similar objective, none of the mentioned approaches implement façade-based data access. That is, none of the existing methods and tools provide Semantic Web practitioners with direct access to heterogeneous sources with SPARQL through a homogenous data model.

However, Shex-ML and RML engines as well as SPARQL Generate have similarities with the *sliced* approach presented in this paper. These approaches, as well as our sliced materialised view approach, will only satisfy queries where there are no joins between data in different portions (for example, it would not be possible to compare values from different CSV rows). Instead, a complete materialised view approach enables any type of graph patterns, to be expressed. However, none of the queries in the GTFS benchmark require this feature to return correct answers. We leave the study of how to support such corner cases with large data sources to future work.

Finally, here we focused on the challenge of executing façade-based SPARQL queries over data files specifically, rather then, for instance, improving the efficiency of general knowledge graph construction (KGC) pipelines. Therefore, while we use and extend a KGC benchmark in our evaluation – the GTFS benchmark [10] – we leave a comparative evaluation of the efficiency of the different approaches to future work.

6. Conclusions

In this paper we explore alternative approaches to façade-based data access with SPARQL and study different approaches to materialised views. We demonstrate, through extensive experiments, that façade-based data access can be a sustainable approach also to transform large files and that it is possible to execute queries effectively even in conditions where the size of data sources challenge the available memory. Furthermore, we have shown how complete, in-memory views are preferable in terms of overall time efficiency, and how further optimisations, such as triple-filtering, can contribute to improve the efficiency of the process, in certain query conditions. These findings are going to inform the documentation of SPARQL Anything and sketch the basis for developing an automatic method to switch between in-memory and on-disk methods, in accordance with the resources available. In the future, we plan to further investigate the impact of slicing on the execution time with large data sources without time constraints. Future work includes designing new execution strategies, including materialising intermediate, partial views, or design query-rewriting techniques. In addition, we aim at evolving triple-filtering to only generate the *optimal* set of triples for a given query.

Acknowledgements

The research has received funding from the European Union's Horizon 2020 research and innovation programme through the project SPICE - Social Cohesion, Participation, and Inclusion through Cultural Engagement (Grant Agreement N. 870811), https://spice-h2020.eu, and the project Polifonia: a digital harmoniser of musical cultural heritage (Grant Agreement N. 101004746), https://polifonia-project.eu.

References

[1] Any23 (May 2022), https://any23.apache.org/
[2] Apache Commons CSV (May 2022), https://commons.apache.org/proper/commons-csv/
[3] ARQ - A SPARQL Processor for Jena (May 2022), https://jena.apache.org/documentation/query/
[4] CSV2RDF (May 2022), http://clarkparsia.github.io/csv2rdf/
[5] Jackson (May 2022), https://github.com/FasterXML/jackson
[6] JSON2RDF (May 2022), https://github.com/AtomGraph/JSON2RDF
[7] Arenas-Guerrero, J., Scrocca, M., Iglesias-Molina, A., Toledo, J., Pozo-Gilo, L., Dona, D., Corcho, O., Chaves-Fraga, D.: Knowledge Graph Construction: An ETL System-Based Overview. ESWC 2021 Workshop KGCW (Submission) (2021)
[8] Asprino, L., Daga, E., Gangemi, A., Mulholland, P.: Knowledge graph construction with a façade: A unified method to access heterogeneous data sources on the web. ACM Trans. Internet Technol. (nov 2022). , https://doi.org/10.1145/3555312, just Accepted

[9] Calvanese, D., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., Xiao, G.: Ontop: Answering sparql queries over relational databases. Semantic Web **8**(3), 471–487 (2017)

[10] Chaves-Fraga, D., Priyatna, F., Cimmino, A., Toledo, J., Ruckhaus, E., Corcho, O.: GTFS-Madrid-Bench: A benchmark for virtual knowledge graph access in the transport domain. Journal of Web Semantics **65**, 100596 (2020)

[11] Chaves-Fraga, D., Ruckhaus, E., Priyatna, F., Vidal, M.E., Corcho, O.: Enhancing virtual ontology based access over tabular data with morph-csv. Semantic Web **12**(6), 869–902 (2021)

[12] Chiatti, A., Daga, E.: Neuro-symbolic learning for dealing with sparsity in cultural heritage image archives: an empirical journey (2022)

[13] Cyganiak, R.: Tarql (sparql for tables): Turn csv into rdf using sparql syntax. Tech. rep., Technical Report, 2015. Available at: http://tarql.github.io (2015)

[14] Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C (Feb 2014), https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

[15] Daga, E., Asprino, L., Mulholland, P., Gangemi, A.: Facade-X: an opinionated approach to SPARQL anything. Studies on the Semantic Web **53**, 58–73 (2021)

[16] Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: Rml: a generic language for integrated rdf mappings of heterogeneous data. In: Proc of LDOW (2014)

[17] García-González, H., Boneva, I., Staworko, S., Labra-Gayo, J.E., Lovelle, J.M.C.: Shexml: improving the usability of heterogeneous data mapping languages for first-time users. PeerJ Computer Science **6**, e318 (2020)

[18] Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C recommendation, W3C (Mar 2013), https://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[19] Johnson, R., Vlissides, J.: Design patterns. Elements of Reusable Object-Oriented Software Addison-Wesley, Reading (1995)

[20] Jozashoori, S., Vidal, M.E.: Mapsdi: A scaled-up semantic data integration framework for knowledge graph creation. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". pp. 58–75. Springer (2019)

[21] Lefrançois, M., Zimmermann, A., Bakerally, N.: A sparql extension for generating rdf from heterogeneous formats. In: Proc of ESWC. pp. 35–50. Springer (2017)

[22] Michel, F., Faron-Zucker, C., Gandon, F.: SPARQL microservices: lightweight integration of web APIs and linked data. In: LDOW@ WWW (2018)

[23] Nuzzolese, A.G., Gangemi, A., Presutti, V., Ciancarini, P.: Fine-tuning triplification with semion. Proc of KIELD pp. 2–14 (2010)

[24] Ratta, M., Daga, E.: Knowledge graph construction from musicxml: An empirical investigation with sparql anything. Proceedings of the first workshop on Musical Heritage Knowledge Graphs (MHKG), co-located with the 21st International Semantic Web Conference (ISWC)

[25] Warren, P., Mulholland, P.: Using SPARQL–the practitioners' viewpoint. In: European Knowledge Acquisition Workshop. pp. 485–500. Springer (2018)