Compiling SHACL Into SQL

Compiling SHACL into SQL

Maxime Jakubowski[1][0000−0002−7420−1337] and Jan Van den Bussche[1][0000−0003−0072−3252]

Data Science Institute, Universiteit Hasselt
https://mjakubowski.info
https://vdbuss.github.io

Abstract. Constraints on graph data expressed in the Shapes Constraint Language (SHACL) can be quite complex. This brings the challenge of efficient validation of complex SHACL constraints on graph data. This challenge is remarkably similar to the processing of analytical queries, investigated intensively in the database community. Motivated by this observation, we have devised an efficient compilation technique from SHACL into SQL, under a natural relational representation of RDF graphs. Our conclusion is that the powerful processing and optimization techniques, already offered by modern SQL engines, are more than up to the challenge.

1 Introduction

The Shapes Constraint Language (SHACL) is a W3C recommended language for expressing integrity constraints on RDF graphs [37,29,45]. In this setting, a "shape" is a possibly complex condition on nodes in a graph; intuitively, a shape specifies how the "neighborhood" of a node in the graph should look like.[1] A SHACL document, called a *shapes graph*, contains various shape definitions, along with target expressions that specify simple node-selecting queries. An RDF graph G *conforms* to a shapes graph S if for every shape–target pair (σ, τ) in S, all nodes in G selected by τ satisfy σ in G. The task of checking whether a graph conforms to a shapes graph (and reporting the violations, if any) is called validation.

Our point of departure in this paper is that validating an RDF graph is strikingly similar to querying a database. Indeed, looking for violations to a shape–target pair entails computing the set difference $Q_1 - Q_2$ between two queries: Q_1 finds all nodes selected by the target, and Q_2 finds all nodes satisfying the shape. A crucial observation in this respect is that SHACL is a rich logical language. Its expressive power certainly includes set difference, through the logic primitives `sh:and`, `sh:not`. Moreover, target expressions correspond themselves to simple shapes. Hence, the difference $Q_1 - Q_2$ above can itself be seen as a shape. Furthermore, finding the violations to all shape–target pairs listed in the

[1] An alternative, automata-based, but equally interesting and popular approach to shapes is taken in the language ShEx [55,14,29].

shapes graph, boils down to finding the set union (logical primitive `sh:or`) of the violations of all these pairs. We are led to conclude that validation is equivalent to *shape querying:* finding all nodes satisfying some (possibly complex) shape.

Of course, querying graph data, in particular RDF graphs, is what SPARQL engines are all about. Our **hypothesis** in this paper, however, is that shape queries are actually much closer to *analytical queries* in relational databases, than they are to SPARQL queries. Analytical queries are complex, ad-hoc queries for decision support. They are usually non-monotonic, and, expressed in SQL, they involve not only joins (typically along star-shaped schemas) but also aggregations, and nested subqueries, often negated (NOT EXISTS in SQL). Queries of this nature are exemplified in the widely used TPC-H benchmark [57].[2]

When analyzing various features by which SHACL can express shapes, and viewing them from the perspective of SQL query constructions, we are indeed strongly reminded of analytical querying:

Group joins and count aggregation Qualified min- and max-count conditions in SHACL amount to group joins [42,22,26]: grouping combined with subqueries and HAVING conditions on the results of aggregate functions (in this case, count aggregations).

MIN and MAX aggregation These arise in evaluating `sh:lessThan` constraints.

Set difference This quintessential non-monotonic operator is involved in the evaluation of various SHACL constraints, including `sh:closed` constraints; disjointness (`sh:disjoint`); and negated equality (`sh:not sh:equals`).

Nested subqueries Further non-monotonic constraints in SHACL require complex nested NOT-EXISTS subqueries, notably equality (`sh:equals`), and the constraint `sh:uniqueLang`.

One may argue that aggregations and nested subqueries, possibly negated, can be expressed in SPARQL equally well as in SQL [32]. Nevertheless, the performance of SPARQL engines has typically been tuned towards monotonic queries, notably, basic graph patterns and path queries. Indeed, SPARQL engines, using index data structures geared to graph data and multiway join algorithms, can outperform relational systems in this area [33]. In contrast, progress in relational database query processing, query plan selection, and query optimization, has been building up for more than fifty years. Especially in the past twenty years, advances in hardware gave rise to breakthroughs in single-node, main-memory SQL engines for analytical querying.[3] Breakthrough techniques included columnar storage and vectorization, as seen in the seminal systems C-Store and MonetDB [56,13,12], later followed by the technique of query compilation [35,36,38].

[2] In the early days of data analytics, analytical querying was referred to as OLAP and was typically focused on data cube operations [31,43]. Later, the term broadened to complex SQL querying; it is in this sense that we use the term "analytical query" in the present paper.

[3] We are leaving as out of scope a discussion of the advances in cloud database systems.

Our goal in this paper is to investigate if our research hypothesis can be tested "straight out of the box". Thereto, we have devised a translation method from SHACL shapes Q into SQL select-statements Q'. The translation is *correct* in the sense that, for any RDF graph G, the result of Q' on a relational database representation of G consists precisely of all nodes in G that satisfy Q. Validation of an entire shapes graph can then be done by looking for violations, as already discussed above. The relational representation of RDF graphs that we assume for our translation is standard [23] and not optimized in any way.

We have taken care to obtain a translation that is also *efficient*. Since SQL is meant to be a declarative language, the notion of "efficient" or "less efficient" SQL expressions is a bit of an oxymoron. Indeed, through query plan selection and optimization, query compilers are ideally supposed to pick a good execution plan regardless of how the query was formulated. Yet, in practice, it makes sense to generate SQL expressions from which the query optimizer is likely to pick a good plan. We achieve this by converting SHACL shapes in *negation normal form* by pushing all negations (`sh:not`) through until they apply to atomic constraint components. We generate a concise SQL expression for each atomic or negated atomic constraint component. Notably, NOT EXISTS subqueries are only needed in that step; SQL expressions for more complex shapes can now be built up using join, grouping and aggregation, filtering, union and set difference operators.

With this translation in hand, all we need to answer our "out of the box" question is a standalone main-memory SQL engine good in processing analytical queries, much like we have standalone main-memory SHACL validators [53]. We have chosen DuckDB [47] for this purpose, as it is very easy to use and install.[4] Our experimental results show that, when validating moderate numbers of target nodes in large graphs, SQL performs equally well as a specialized SHACL engine. When validating a substantial proportion of nodes, shape querying using SQL becomes orders of magnitude more efficient. We also compare with Trav-SHACL, which is a recent approach to SHACL processing through SPARQL rather than through SQL [27].

This paper is organized as follows. Section 2 offers some further motivation for shape querying. Section 3 discusses related work. Section 4 recalls the SHACL language, using a logical syntax. Section 5 presents the translation into SQL. Section 6 presents experiments showing the viability of our approach. Section 7 draws conclusions.

2 Motivation

Shape querying is also motivated by new applications of shapes, going beyond validation. Shape fragments [21] and knowledge-graph subsets [39] take shape querying one step further: they retrieve not just all nodes satisfying some shape, but also the "neighborhoods" of these nodes. The neighborhoods depend on the

[4] Alternatively we could have used the Hyper API `https://www.tableau.com/developer/tools/hyper-api`.

shape. In this way, shapes can be used as a view mechanism. Delva et al. [21] use a SHACL2SPARQL-like translation as an aid to generate even more complex SPARQL queries that return the neighborhoods. They provide little detail concerning their translation, its efficiency, or comparison to other systems.

Views defined by shapes may also have application in access control. In current approaches [59,51], the credentials of a user are checked by validating a SHACL shapes graph against RDF data containing information about this user. In case of conformance, the user is granted access to a data source according to one of the standard access control levels (e.g., read, write, etc.) However, shapes could also be used to specify, in a more fine-grained manner, a *subgraph* of the source to which access is granted [20].

3 Related work

The idea to reduce SHACL validation to querying comes naturally. Indeed, while we take an approach through SQL in this paper, Corman et al. [16] already investigated a more direct approach through SPARQL. While their focus is mostly on recursion, they also considered the translation of a nonrecursive shapes graph into a single complex SPARQL query. They also introduce a rule-based algorithm that can handle recursion. The single-query approach was reported to perform slightly better than the rule-base algorithm in the nonrecursive case. They do not compare performance with other SHACL engines. Their system, SHACL2SPARQL, supports only a limited class of shapes [54].

Trav-SHACL [27] was introduced as an improvement over SHACL2SPARQL; we include Trav-SHACL in our experiments in Section 6. Unfortunately, also Trav-SHACL currently only supports a quite limited set of constraint components. Our translation to SQL presented in this paper covers all SHACL core constraint components, but omits property paths, as explained in Section 4.

Representing RDF graphs in relational databases, which is necessary for translating SHACL to SQL, is an old idea dating back to at least the Virtuoso system [25]. Conversely, while research on implementation and optimization of analytical queries has been predominantly pursued in the relational setting (cf. the references in the Introduction), such techniques can certainly also be integrated in RDF systems [44,24,11,2,34]. The Virtuoso system is rumored to have some SHACL functionality, but, to the best of our knowledge, there is no detailed information available. Searches in the docs of both commercial and open-source software returned no results.

To conclude we mention existing work which relates SHACL and SPARQL on a quite different level than considered here [1,48,9]. There, the goal is not to do SHACL processing by SPARQL querying, but rather, to use knowledge about integrity constraints expressed in SHACL to optimize SPARQL queries posed against conforming data.

Recursion Many interesting research works on SHACL are focused on recursive shapes graphs, e.g., [17,7,16,46,10,15,3,4,5,6]. Indeed, one sometimes gets the

feeling that some authors seem to suggest that SHACL without recursion is trivial. We strongly object to this conception. Our work shows that nonrecursive shapes graphs already give rise to complex SQL queries that no database expert would call trivial. This does not mean, though, that the extension of our work to recursion, e.g., by using recursive SQL or SQL stored procedures, would not be a good topic for further work.

4 Preliminaries on SHACL

SHACL has been defined with an RDF syntax. As in many research works around SHACL, however [17,7,16,40,3,4,5,6], we use here a *logical syntax* instead. We follow the most complete proposal [21] which covers the entire SHACL core, with some slight alterations.

Assume three infinite pairwise-disjoint sets I, L and B of IRIs, literals and blank nodes respectively. We call the union of these sets $N = I \cup L \cup B$ the set of RDF terms. Literals generally have three attributes [49]: a value, a datatype and a language tag. For the purposes of SHACL, we also need to assume a strict partial order $<$ on N as an abstraction of comparisons between RDF terms.[5]

An *RDF triple* (s, p, o) is an element of $(I \cup B \times I \times N)$. We refer to the elements of the triple as the subject, predicate and object respectively. An *RDF graph* G is a finite set of RDF triples. We refer to the RDF terms occurring in the subject or object positions of an RDF graph as *nodes*.

SHACL shapes can make use of regular path expressions, called *property paths*, also known from SPARQL. Efficient regular path queries require specialized techniques [8] going beyond SQL, even recursive SQL [19]. In the present paper we omit them from our treatment. We thus arrive at the syntax for *shape expression* φ as given by the following grammar:

$$
\begin{aligned}
E &::= p \mid p^- \\
F &::= id \mid E \\
\varphi &::= \top \mid hasShape(s) \mid hasValue(c) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \\
&\quad \mid \#_n^m E.\varphi \mid \forall E.\varphi \mid eq(F,p) \mid disj(F,p) \mid closed(P) \\
&\quad \mid lessThan(E,p) \mid lessThanEq(E,p) \mid uniqueLang(E) \mid test(t)
\end{aligned}
$$

with $p \in I$; $s \in I \cup B$; $c \in N$; n a natural number; m is either a natural number or the symbol '$*$'; and $P \subseteq I$ finite. Here E represents a limited form of a path expression. SHACL supports testing whether nodes satisfy certain properties. We abstract this with the $test(t)$ feature where t represents a well-defined node-test. Examples of node tests are `sh:nodeKind`, testing whether a node is an IRI, blank node or literal; or `sh:languagein`, testing whether a node has one of the specified language tags. The specific allowed tests are discussed in Section 5. We will sometimes use shape expressions of the form $\exists E.\varphi$ to denote $\#_1^* E.\varphi$.

[5] This operator is defined by the Operator Mapping in SPARQL [32].

Generally, a shape has an associated *shape name* and possibly a *target declaration*. We formalize this notion of a shape as a *shape definition*, which is a triple (s, φ, τ) where $s \in I \cup B$; φ is a shape expression; and τ is a *target declaration*. The latter are specific shapes of one of the four forms: $hasValue(c)$, $\exists p.\top$, $\exists p^-.\top$, and $\exists \texttt{rdf:type}.hasValue(c)$.[6] These four forms correspond to the target declarations `sh:targetNode` c, `sh:targetSubjectsOf` p, `sh:targetObjectsOf` p, and `sh:targetClass` c respectively. When no target declaration is desired, one can use the shape expression $\neg\top$. Finally, we formalize a SHACL shapes graph as a *schema*. A schema is a finite set of shape definitions where no two definitions have the same shape name. In this work, we only consider nonrecursive schemas.

We will now define when a node a conforms to a shape φ in graph G, within the context of a schema H, denoted by $H, G, a \models \varphi$. First, the evaluation of a path expression E, written as $[\![E]\!]^G$ is given as follows. Let $p \in I$: $[\![p]\!]^G = \{(a, b) \mid (a, p, b) \in G\}$ and $[\![p^-]\!]^G = \{(a, b) \mid (b, a) \in [\![p]\!]^G\}$. Then, the semantics of shape expressions is given in Table 1.

We use the following notations:

- $def(s, H)$ denotes the shape expression defining shape name s in H. When s does not have a definition in H, $def(s, H) = \top$ (which is the behaviour of real SHACL).
- We use the notation $[\![E]\!]^G(a)$ to denote the set $\{b \mid (a, b) \in [\![E]\!]^G\}$.
- When X is a set, we use the notation $\sharp X$ to denote the cardinality of X.

Table 1. Conditions for conformance of a node to a shape.

φ	$H, G, a \models \varphi$ if:
$hasValue(c)$	$a = c$
$test(t)$	a satisfies t
$hasShape(s)$	$H, G, a \models def(s, H)$
$\#_n^m E.\psi$	$\begin{cases} n \leq \sharp\{b \in [\![E]\!]^G(a) \mid H, G, b \models \psi\} \leq m & \text{if } m \neq * \\ n \leq \sharp\{b \in [\![E]\!]^G(a) \mid H, G, b \models \psi\} & \text{otherwise} \end{cases}$
$\forall E.\psi$	every $b \in [\![E]\!]^G(a)$ satisfies $H, G, b \models \psi$
$eq(F, p)$	the sets $[\![F]\!]^G(a)$ and $[\![p]\!]^G(a)$ are equal
$disj(F, p)$	the sets $[\![F]\!]^G(a)$ and $[\![p]\!]^G(a)$ are disjoint
$closed(P)$	for all triples $(a, p, b) \in G$ we have $p \in P$
$lessThan(E, p)$	$b < c$ for all $b \in [\![E]\!]^G(a)$ and $c \in [\![p]\!]^G(a)$
$lessThanEq(E, p)$	$b \leq c$ for all $b \in [\![E]\!]^G(a)$ and $c \in [\![p]\!]^G(a)$
$uniqueLang(E)$	for every $b \neq c \in [\![E]\!]^G(a)$, b and c have different language tags.

[6] Actually, in the case of targetclass, we simplify to only targeting direct type declarations.

In general, it is easy to see that if a shape expression does not refer to other shapes, we do not need to include the schema as part of the conformance definition: in that case $G, a \models \varphi$ is well defined.

Finally, we can define when a graph conforms to a schema. An RDF graph G conforms to a shape schema H if for every shape definition $(s, \varphi, \tau) \in H$ and for every $a \in N$ such that $H, G, a \models \tau$ we have $H, G, a \models \varphi$.

Since we focus on nonrecursive schemas, we can, for the purposes of conformance, abstract away the $hasShape(s)$ construct from the shape expression grammar because we can always *expand* the shape expression with the definition of that shape name, resulting in a semantically equivalent shape expression. We define the *expansion* of a shape expression φ in context of a schema H as the shape expression ψ which replaces every shape name s occurring in the construct $hasShape(s)$ in φ with $def(s, H)$ resulting in the expression φ'. We inductively apply this principle to φ' until we end up with a shape expression that does not refer to any other shape definitions.

As mentioned in the Introduction, we leverage the *negation normal form* [50] of a shape expression to acquire an efficient translation. The negation normal form of a shape expression φ is the shape expression φ' in which we push every negation of a subshape through. We define the negation normal form by giving the rewrite rules that need to be performed on φ to obtain φ':

- $\neg(\varphi_1 \wedge \varphi_2)$ becomes $\neg\varphi_1 \vee \neg\varphi_2$;
- $\neg(\varphi_1 \vee \varphi_2)$ becomes $\neg\varphi_1 \wedge \neg\varphi_2$;
- $\neg\neg\varphi$ becomes φ;
- $\neg\forall E.\varphi$ becomes $\exists E.\neg\varphi$;
- when $m \neq *$ and $n \neq 0$, $\neg\#_n^m E.\varphi$ becomes $\#_0^{n-1} E.\varphi \vee \#_{m+1}^* E.\varphi$;
- when $m \neq *$ and $n = 0$, $\neg\#_n^m E.\varphi$ becomes $\#_{m+1}^* E.\varphi$; and
- when $m = *$ and $n \neq 0$, $\neg\#_n^m E.\varphi$ becomes $\#_0^{n-1} E.\varphi$.

5 Translation

Our translation covers all the SHACL-core features formalized in the previous section. The only technical limitation is that we only support datatype-comparisons for numerical values and strings. For example, we do not support `sh:minInclusive` tests on literals that represent dates.

In order give this translation, we first need to describe the relational database schema that represents the underlying RDF data. The relational schema described here tries to be very close to the definition of the RDF data model. We adopt a standard technique "pooling" technique which associates a unique identifier to all RDF nodes, which is then used in the central Triples relation. The full schema is given by the following relations:

IRIs(Node: int64, Value: string) This relation stores all nodes that are IRIs. The `Value` attribute stores the IRI as a string, and relates it with a unique node identifier given by the `Node` attribute.

Blanks(Node: int64, Alias: string) Similarly to the IRI table, we keep track of blank nodes by associating an identifier with them.

Literals(Node: int64, Value: string, Type: string, Lang: string) Again, similar to the IRIs table, we keep track of the literals used in the graph. Furthermore, it relates node identifiers to the value that they represent, the type that they are designated and the language tag that may be present. The language tag field is not NULL precisely when the datatype attribute value is `rdf:langString`, as expected.

Nodes(Node: int64) This relation stores all node identifiers.

Triples(Subject: int64, Predicate: string, Object: int64) This is the central relation of our schema. The `Subject` and `Object` attributes refer to the node identifier used in the `Node` attribute of one of the previous three relations. We do not need to use node ids in the predicate column, since property names cannot be treated as nodes in SHACL.

Numerics(Node: int64, Value: double) The primary purpose of this relation is to have quick access to a numeric (double) value associated with literals that represent numerics. The `Node` column is a subset of the `Node` column in the `Literals` table. This relation is used to support some of the node-tests that check for numeric constraints, like `sh:lessThan` of `sh:minExclusive`.

In general, given an RDF graph G and a shape expression φ that does not refer to other shapes, our translation gives us a unary SQL query Q_φ that retrieves all nodes in G that satisfy φ. In the case of shapes that *do* refer to other shapes in context of a schema H, we can also translate the shape using the same techniques, by first expanding the shape expression with respect to H.

We will now give the translation for shape expressions in negation normal form. For now, we leave out the details for supporting inverse properties. Generally, when p is inverse, we need to swap the operations relating to the Subject and Object columns of the Triples relation. For shape expressions φ of the form $\varphi_1 \wedge \varphi_2$, query Q_φ is `(` Q_{φ_1} `) INTERSECT (` Q_{φ_2} `)`. For shape expressions φ of the form $\varphi_1 \vee \varphi_2$, query Q_φ is `(` Q_{φ_1} `) UNION (` Q_{φ_2} `)`. Next, for the counting shapes of the form $\#_n^m p.\psi$, we consider four cases. First, when $n > 0$, $m \neq *$, and $n \neq m$, query Q_φ is

```
SELECT Subject AS Node FROM Triples, (Q(ψ)) AS Q(Node)
WHERE Predicate = p AND Object = Q.Node
GROUP BY Subject
HAVING COUNT(*) >= n AND COUNT(*) <= m
```

When $n = m$, we replace the last line with `HAVING COUNT(*) =` n; when $m = *$ (and thus $n > 0$), we replace it with `HAVING COUNT(*) >=` n. When $n = 0$ (and thus $m \neq *$), we instead have the query:

`SELECT Node FROM Nodes WHERE Node NOT IN (`$Q_{\#^*_{m+1} p.\psi)}$`)`

Indeed, we cannot simply use `HAVING COUNT(*) >= 0` in this case (compare the infamous "count bug" [28].) Lastly, when the subshape ψ is simply $\top$, we can leave out the subquery: `(`$Q(\psi)$`) AS Q(Node)`.

Next, for the case where φ is of the form $\forall p.\psi$ we have the query:

```
SELECT Node FROM Nodes WHERE NOT EXISTS (
  SELECT * FROM Triples, (Q_ψ) AS Q(Node)
  WHERE Predicate = p AND Subject = Node AND Object NOT IN Q )
```

The remaining non-test shapes are listed in Table 2. Before discussing the test shapes, we will list the allowed node-tests in SHACL. These are: $nodeKind(X)$ with $X \in \{i, b, l\}$ representing the test whether a node is an IRI, blank node or literal; $datatype(d)$ with $d \in I$ checking whether a node has a certain datatype, e.g., `xsd:integer`; $minIncl(n)$, $minExcl(n)$, $maxIncl(n)$, $maxExcl(n)$ which represent the *value range constraint components* of SHACL [37]; $minLength(n)$, $maxLength(n)$ which state that the string representation of literals must have a minimal, maximal length; $pattern(p, f)$ state that the string representation of literals must satisfy some regular expression p (with flags f); and finally $languagein(L)$ which states that the literal must have one of the language tags from the set L.

The test shapes are listed in Table 3 with the exception of numeric and string length constraints. For the numeric length constraints, like $minExcl(n)$ we get the following query: `SELECT Node FROM Numerics WHERE Value >` n. It is clear what queries we get for every one of its variations; we simply need to change the `>` operator: $minIncl(n)$ (inclusive, `>=`), $maxExcl(n)$ (max exclusive `<`), $maxIncl(n)$ (max inclusive `<=`). When combinations of these tests are used, we can simply add boolean combinations in the where-clause, for example $minIncl(n) \wedge maxExcl(m)$ becomes: `SELECT Node FROM Numerics WHERE Value <=` n `AND Value >` m.

Similar techniques are applied to the string length tests $minLength(n)$ and $maxLength(n)$.

Table 2: Translation of shape expressions to unary SQL queries. We omit the case where φ is $\top$.

φ	Q_φ	$Q_{\neg\varphi}$

$disj(p, q)$	`SELECT Node`
	`FROM Nodes`
	`EXCEPT (` $Q_{\neg disj(p,q)}$ `)`
		`SELECT T1.Subject AS Node`
		`FROM Triples AS T1,`
		`Triples AS T2`
		`WHERE T1.Subject = T2.Subject`
		`AND T1.Predicate =` p
		`AND T2.Predicate =` q
		`AND T1.Object = T2.Object`

$disj(id, p)$	`SELECT Node`
	`FROM Nodes`
	`EXCEPT (` $Q_{\neg disj(id,p)}$ `)`
		`SELECT Subject AS NODE`
		`FROM Triples`
		`WHERE Predicate =` p
		`AND Subject = Object`

$eq(p, q)$

```
SELECT Node FROM Nodes
WHERE NOT EXISTS (
(( SELECT Object
   FROM Triples
   WHERE Predicate = p
     AND Subject = Node
 ) EXCEPT (
   SELECT Object
   FROM Triples
   WHERE Predicate = q
     AND Subject = Node
 )) UNION ((
   SELECT Object
   FROM Triples
   WHERE Predicate = q
     AND Subject = Node
  ) EXCEPT (
   SELECT Object
   FROM Triples
   WHERE Predicate = p
     AND Subject = Node )))
```

```
SELECT Node FROM Nodes
WHERE EXISTS ((
 SELECT * FROM Triples
 WHERE Predicate = p
  AND Object NOT IN (
   SELECT Object From Triples
   WHERE Subject = Node
    AND Predicate = q )
  ) UNION (
 SELECT * FROM Triples
 WHERE Predicate = q
  AND Object NOT IN (
   SELECT Object From Triples
   WHERE Subject = Node
    AND Predicate = p )))
```

$eq(id, p)$

```
SELECT Subject AS Node
FROM Triples AS T1
WHERE Predicate = p
  AND Subject = Object
  AND NOT EXISTS (
SELECT *
FROM Triples AS T2
WHERE Predicate = p
  AND T2.Subject = T1.Subject
  AND T2.Object <> T1.Object )
```

```
SELECT Node FROM Nodes
WHERE Node NOT IN (
  SELECT * FROM Triples
  WHERE Subject = Node
    AND Predicate = p
  ) OR EXISTS (
SELECT * FROM Triples
WHERE Predicate = p
  AND Object <> Node )
```

$closed(P)$

```
SELECT Node
FROM Nodes
EXCEPT ( Q_{¬closed(P)} )
```

```
SELECT Subjects AS Node
FROM Triples
WHERE
 Predicate NOT IN (P) )
```

$lessThan(p, q)$	SELECT Node
	FROM Nodes WHERE (
	  SELECT MAX(N.Value)
	  FROM Triples AS T,
	  Numerics AS N
	  WHERE T.Predicate = p
	  AND T.Subject = Nodes.Node
	  AND T.Object = N.Node
	) < (
	SELECT MIN(N.Value)
	FROM Triples AS T,
	  Numerics AS N
	  WHERE T.Predicate = q
	  AND T.Subject = Nodes.Node
	  AND T.Object = N.Node)
		SELECT T1.Subject AS Node
		FROM Triples AS T1,
		  Triples AS T2,
		  Numerics AS N1,
		  Numerics AS N2
		WHERE T1.Predicate = p
		  AND T2.Predicate = q
		  AND T1.Object = N1.Node
		  AND T2.Object = N2.Node
		  AND N1.Value >= N2.Value

$uniqueLang(p)$	SELECT Node FROM Nodes
	WHERE NOT EXISTS (
	  SELECT L.Lang
	  FROM Triples AS T,
	  Literals AS L
	  WHERE T.Predicate = p
	  AND T.Subject = Nodes.Node
	  AND T.Object = L.Node
	  AND L.Lang NOT NULL
	  GROUP BY L.Lang
	  HAVING COUNT(*) > 1)
		SELECT Subject AS Node
		FROM Triples AS T1,
		  Tripples AS T2,
		  Literals AS L1,
		  Literals AS L2
		WHERE T1.Subject = T2.Subject
		  AND T1.Predicate = p
		  AND T2.Predicate = p
		  AND T1.Object = L1.Node
		  AND T2.Object = L2.Node
		  AND L1.Lang <> L2.Lang

6 Experiments

We implemented the translation from SHACL to SQL in Python, using the popular library RDFLib. Our implementation can translate real SHACL shapes graphs into SQL queries that retrieve all nodes satisfying a shape expression, or all violations of a shape definition. It supports all features discussed in the previous section. To load RDF graphs into a DuckDB database, we also wrote a simple translation tool. DuckDB automatically creates min-max indexes for all table columns.

We validated the correctness of our implementation by running it against the SHACL core test suite. Our implementation passed 70 percent of the core tests. The non-passed tests either have to do with property paths, or with tests on datatypes as already mentioned in the beginning of Section 5.

We compare our SQL approach to two other SHACL engines: the Apache Jena SHACL validator, as a representative of a dedicated SHACL engine, and Trav-SHACL, as a representative of the SHACL-to-SPARQL approach [58]. Our experiments measure the execution time of validating a shape against a graph. For the SQL approach, we measure the query execution time of DuckDB. For Apache Jena we first parse the shape schema and load the data into a TDB

Table 3. Translation of shape expressions to SQL for the *hasValue* and *test* features. In this table, $c_i \in I$, $c_l \in L$. We omit $nodeKind(b)$ and $nodeKind(l)$.

φ	Q_φ	$Q_{\neg\varphi}$

$hasValue(c_i)$	`SELECT Node`
	`FROM IRIs`
	`WHERE Value =` c_i
		`SELECT Node FROM Literals`
		`UNION`
		`SELECT Node FROM Blanks`
		`UNION`
		`SELECT Node FROM IRIs`
		`WHERE Value <>` c_i

$hasValue(c_l)$	`SELECT Node`
	`FROM Literals`
	`WHERE Value =` $c_l.value$
	`AND Type =` $c_l.datatype$
	`AND Lang =` $c_l.language$
		`SELECT Node FROM IRIs`
		`UNION`
		`SELECT Node FROM Blanks`
		`UNION`
		`SELECT Node FROM Literals`
		`WHERE Value <>` $c_l.value$
		`OR Type <>` $c_l.datatype$
		`OR Lang <>` $c_l.language$

$hasValue(b)$	$Q_{\neg\top}$	$Q_\top$

$nodeKind(i)$	`SELECT Node FROM IRIs`	`SELECT Node FROM Blanks`
		`UNION`
		`SELECT Node FROM Literals`

$datatype(d)$	`SELECT Node`
	`FROM Literals`
	`WHERE Datatype =` d
		`SELECT Node FROM IRIs`
		`UNION`
		`SELECT Node FROM Blanks`
		`UNION`
		`SELECT Node FROM Literals`
		`WHERE Datatype <>` d

$pattern(p, f)$	`SELECT Node FROM Literals`
	`WHERE regex(Value,`p`,`f`)`
	`UNION`
	`SELECT Node FROM IRIs`
	`WHERE regex(Value,`p`,`f`)`
		`SELECT Node FROM Blanks`
		`UNION`
		`SELECT Node FROM Literals`
		`WHERE NOT regex(Value,`p`,`f`)`
		`UNION`
		`SELECT Node FROM IRIs`
		`WHERE NOT regex(Value,`p`,`f`)`

$languagein(L)$	`SELECT Node FROM Literals`
	`WHERE Language IN (`L`)`
		`SELECT Node FROM IRIs`
		`UNION`
		`SELECT Node FROM Blanks`
		`UNION`
		`SELECT Node FROM Literals`
		`WHERE Language NOT IN (`L`)`
		`OR Language IS NULL`

database, we then measure the execution time of the `ShaclValidator.validate` function. For Trav-SHACL, we first setup an Jena Fuseki SPARQL endpoint exposing the data, and parse the shape schema. We then measure the execution time of the `ShapeSchema.validate` function.

Other validation engines were also considered. PySHACL requires the graph to be loaded in main memory, and ran out of memory for most of our datasets. We also observed that TopQuadrant's engine performs very similarly to Jena, so we omit TopQuadrant from the presentation of our results to avoid clutter.

We used a 8 core AMD EPYC 2.595GHz processor with 16GB DDR4 RAM and 400GB SSD to run all experiments.

Synthetic shapes and data As a starting point, we formulated 10 SHACL shapes, shown in Table 4. These shapes were purposely invented to explore a variety of SHACL features. For each shape, we generate suitable synthetic data. For datasets of 5 million triples, we obtain the timings represented in Figure 1 (a), logarithmic scale. Some of the shapes are not supported by Trav-SHACL. We can see that DuckDB is an order of magnitude faster on most shapes, except, curiously, Shape 7.

Table 4. The 10 synthetic shape definitions. All shapes target nodes of type 'human', except for shape 7, which targets all objects of the 'email' property.

Name	Expression
Shape 1	$\exists phone.\top \wedge \neg\exists email.\top$
Shape 2	$\#^{*}_{5} managedBy.\top$
Shape 3	$\exists friend.\exists ceoOf.hasValue(company1)$
Shape 4	$\neg disj(colleague, friend)$
Shape 5	$closed(property1, property2, property3)$
Shape 6	$\exists phone.\top \vee \exists email.\top$
Shape 7	$\#^{1}_{0} email^{-}.\top$
Shape 8	$eq(property1, property2)$
Shape 9	$uniqueLang(firstName)$
Shape 10	$lessThan(startWork, endWork)$

To show that the results of this synthetic experiment scale, we generated datasets of 20 million triples and ran the same experiments with DuckDB and Jena SHACL (not with Trav-SHACL, due to memory limitations). These results can be found in Figure 1 (b), also logarithmic scale.

Remark 1. In Figure 1, we see that for the synthetic experiments shape 7, Jena outperforms DuckDB. We reused the dataset for shape 6 to also test shape 7. However, this dataset has very few targets for shape 7. DuckDB retrieves many conforming nodes but subtracts these from the small list of targets. It still does this in 1 second, but Jena just looks at the very few targets. We already noted in the Tyrol experiments that small target sizes skew the comparison. When

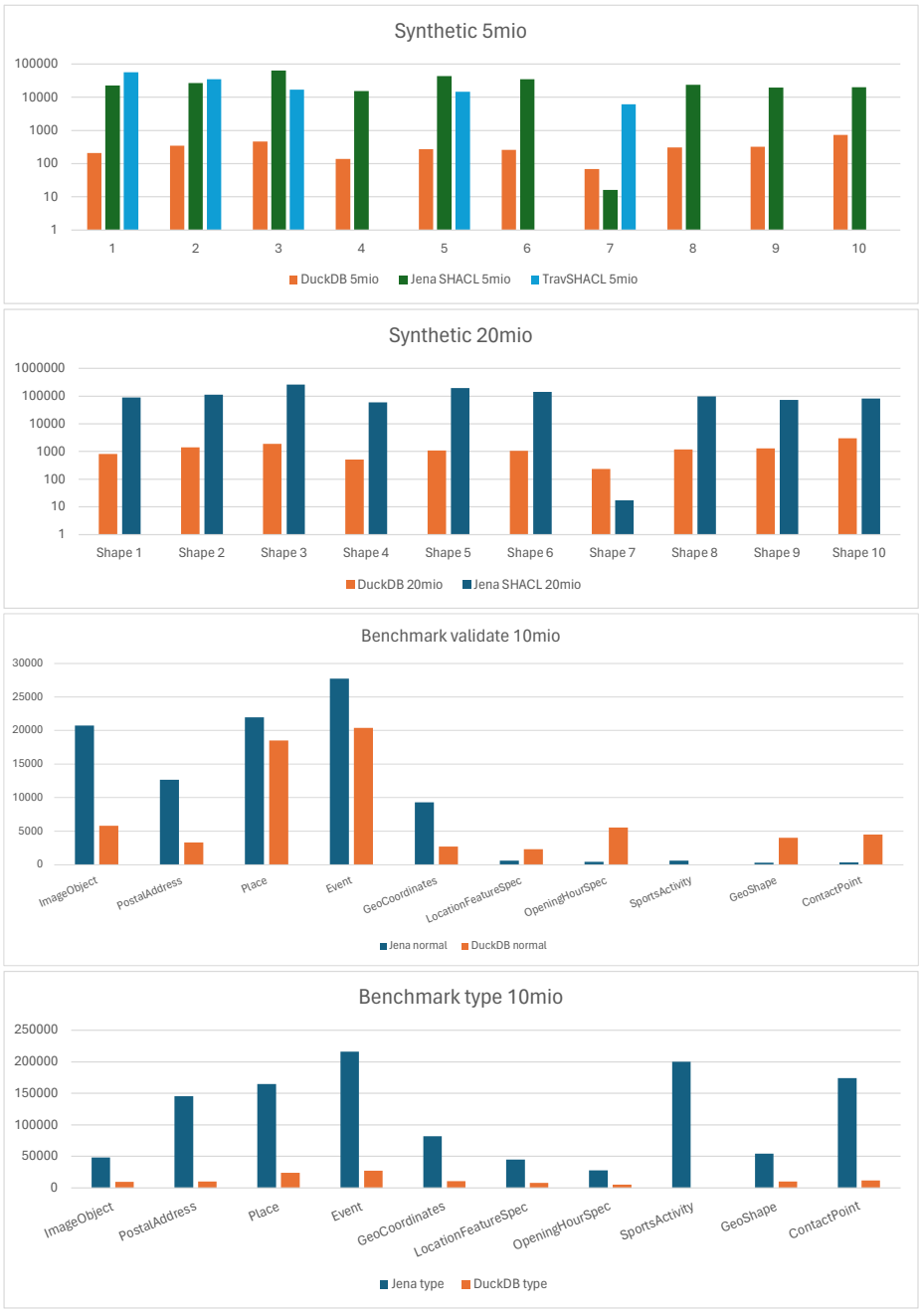

Fig. 1. Execution time in milliseconds for the synthetic and benchmark experiments. From top to bottom, these figures are referred to as (a), (b), (c) and (d).

rerunning the experiments on a more balanced dataset, the quirk disappears. The additional balanced dataset can be found in the supplementary material.

Tyrol benchmark We also ran a selection of shapes from the SHACL benchmark by Schaffenrath et al. [52]. They define 57 shapes over the "Tyrolean Knowledge Graph" which consists of 30 million triples. We selected a random 10 million triple slice of this graph, and selected 10 of the 57 shapes that had the most targets in our slice. We then ran our SQL approach and Jena on this slice. We had to leave out Trav-SHACL, since its current implementation is still rather limited and does not support the features used in the selected shapes.

The performance results are shown in Figure 1 (c). The benchmark shapes are ordered left to right from most to fewest number of targets. We can see that Jena SHACL outperforms our SQL approach for half of the shapes. This can be explained by the fact that the last five shapes have significantly less target nodes to check, compared to the first five shapes. Jena SHACL is a specialized SHACL engine, and can quickly retrieve these targets and perform the validation. However, if we adjust the shapes such that they target *all* subjects of `rdf:type`, which lies closer to shape querying, Jena loses this advantage. This is illustrated in Figure 1 (d). We report that DuckDB ran out of memory for 'SportsActivityShape'. This may be due to the large number of constraints used by this shape.

One major difference between the synthetic data and the benchmark data is that in the synthetic experiments almost all of the data needs to be checked to decide conformance. Here, only small parts of the data needs to be checked for some shapes. This explains why the execution times are generally lower.

DBLP data DBLP published their database as a large RDF graph containing 400 million triples [18]. This RDF graph contains information about publications and their authors. We created three analytical shapes for this dataset:

PersonShape: the shape expression $\#_1^1 p.\top \wedge \neg disj(p, a)$, with p representing primary affiliation and a affiliation. It is run against all nodes of type 'Person' (which represent authors).

TeamplayerShape: the shape expression $\exists a^-.\#_3^* a.\top$, with a representing 'authored by', again run against all authors.

PublicationShape: of a less analytical nature, this is the shape expression $\exists$`dblp:authoredBy`.$\exists$`rdf:type`.$hasValue$(`dblp:Person`), run against all nodes of type 'Publication'.

We run these shapes on purpose against large sets of targets (all authors, or all publications) to get information on shape querying performance. We used a slice of the DBLP database that contains all publications together with their author information for the years 2022 and 2023. This results in an RDF graph of 8.5 million triples. The results are given in Figure 2. Again, DuckDB significantly outperforms Jena SHACL, also on PublicationShape.

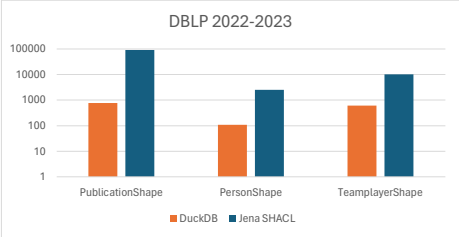

Fig. 2. Execution time in milliseconds for a slice of the DBLP database consisting of 8.5 million triples. Logarithmic scale.

7 Conclusion

We have shown that through a compilation into SQL, shape queries and validations can run on an efficient relational database, without having to make any tweaks. This already significantly outperforms dedicated SHACL engines as well as approaches via SPARQL. Of course, our main message is **not** that RDF data management should move to relational systems. Rather, our message is that the techniques used in these systems, which certainly can be applied in RDF systems as well, will be beneficial for SHACL performance.

It will be interesting to see how usage of SHACL features evolves in practice in the future, e.g., [41]. This will undoubtedly also depend on how fast these features run in RDF systems.

Supplemental Material Statement: For data, shapes and source code, see `https://github.com/MaximeJakubowski/shaclsql-supplementary`.

References

1. Abbas, A., Genevès, P., Roisin, C., Layaïda, N.: Selectivity estimation for SPARQL triple patterns with shape expressions. In: Mikkonen, T., et al. (eds.) Proceedings 18th International Conference on Web Engineering. Lecture Notes in Computer Science, vol. 10845, pp. 195–209. Springer (2018)
2. Ahlstrøm Jakobsen, K., Andersen, A., Hose, K., Bach Pedersen, T.: Optimizing RDF data cubes for efficient processing of analytical queries. In: Hartig, O., Sequeda, J., et al. (eds.) Proceedings 6th International Workshop on Consuming Linked Data. CEUR Workshop Proceedings, vol. 1426 (2015)
3. Ahmetaj, S., David, R., Ortiz, M., Polleres, A., Shehu, B., Simkus, M.: Reasoning about explanations for non-validation in SHACL. In: Bienvenu, M., Lakemeyer, G., et al. (eds.) Proceedings 18th International Conference on Principles of Knowledge Representation and Reasoning. pp. 12–21. IJCAI Organization (2021)
4. Ahmetaj, S., David, R., Polleres, A., Simkus, M.: Reparing SHACL constraint violations using answer set programming. In: Sattler, U., et al. (eds.) Proceedings 21st International Semantic Web Conference. Lecture Notes in Computer Science, vol. 13489, pp. 375–391. Springer (2022)

5. Ahmetaj, S., Löhnert, B., Ortiz, M., Simkus, M.: Magic shapes for SHACL validation. Proceedings of the VLDB Endowment **15**(10), 2284–2296 (2022)
6. Ahmetaj, S., Ortiz, M., Oudshoorn, A., Simkus, M.: Reconciling SHACL and ontologies: Semantics and validation via rewriting. In: Gal, K., Nowé, A., et al. (eds.) Proceedings 26th European Conference on Artificial Intelligence. Frontiers in Artificial Intelligence and Applications, vol. 372, pp. 27–35. IOS Press (2023)
7. Andreşel, M., Corman, J., Ortiz, M., Reutter, J., Savkovic, O., Simkus, M.: Stable model semantics for recursive SHACL. In: Huang, Y., King, I., Liu, T.Y., van Steen, M. (eds.) Proceedings WWW'20. pp. 1570–1580. ACM (2020)
8. Arroyuelo, D., Hogan, A., Navarro, G., Rojas-Ledesma, J.: Time- and space-efficient regular path queries. In: Proceedings 38th International Conference on Data Engineering. pp. 3091–3105. IEEE (2022)
9. Bahadur Thapa, R., Giese, M.: Optimizing SPARQL queries with SHACL. In: Payne, T., Presutti, V., Qi, G., et al. (eds.) Proceedings 22nd International Semantic Web Conference. Lecture Notes in Computer Science, vol. 14265, pp. 41–60. Springer (2023)
10. Bogaerts, B., Jakubowski, M.: Fixpoint semantics for recursive SHACL. In: Formisano, A., Liu, Y., et al. (eds.) Proceedings 37th International Conference on Logic Programming (Technical Communications). Electronic Proceedings in Theoretical Computer Science, vol. 345, pp. 41–47 (2021)
11. Boncz, P., Erling, O., Pham, M.D.: Advances in large-scale RDF data management. In: Auer, S., Bryl, V., Tramp, S. (eds.) Linked Open Data, Lecture Notes in Computer Science, vol. 8661, pp. 21–44. Springer (2014)
12. Boncz, P., Kersten, M., Manegold, S.: Breaking the memory wall in MonetDB. Communications of the ACM **51**(12), 77–85 (2008)
13. Boncz, P., Zukowski, M., Nes, N.: MonetDB/X100: Hyper-pipelining query execution. In: Proceedings 2nd Biennial Conference on Innovative Data Systems Research. pp. 225–237. www.cidrdb.org (2005)
14. Boneva, I., Gayo, J., Prud'hommeaux, E.: Semantics and validation of shape schemas for RDF. In: d'Amato, C., Fernandez, M., Tamma, V., et al. (eds.) Proceedings 16th International Semantic Web Conference. Lecture Notes in Computer Science, vol. 10587, pp. 104–120. Springer (2017)
15. Chmurovic, A., Simkus, M.: Well-founded semantics for recursive SHACL. In: Alviano, M., Pieris, A. (eds.) Datalog 2.0 2022: Fourth International Workshop on the Resurgence of Datalog in Academia and Industry. CEUR Workshop Proceedings, vol. 3203, pp. 2–13 (2022)
16. Corman, J., Florenzano, F., Reutter, J., Savkovic, O.: Validating SHACL constraints over a SPARQL endpoint. In: Ghidini et al. [30], pp. 145–163
17. Corman, J., Reutter, J., Savkovic, O.: Semantics and validation of recursive SHACL. In: Vrandecic, D., et al. (eds.) Proceedings 17th International Semantic Web Conference. Lecture Notes in Computer Science, vol. 11136, pp. 318–336. Springer (2018), extended version, technical report KRDB18-01, `https://www.inf.unibz.it/krdb/tech-reports/`
18. DBLP data in RDF. `http://dblp.org/rdf/`
19. De Leo, D., Boncz, P.: Extending SQL for computing shortest paths. In: Boncz, P., Larriba-Pey, J. (eds.) Proceedings 5th International Workshop on Graph Data-management Experiences & Systems. pp. 10:1–10:8. ACM (2017)
20. Dedecker, R., Slabbinck, W., Wright, J., et al.: What's in a Pod? A knowledge graph interpretation for the Solid ecosystem. In: Saleem, M., et al. (eds.) Proceedings 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs. CEUR Workshop Proceedings, vol. 3279, pp. 81–96 (2022)

21. Delva, T., Dimou, A., Jakubowski, M., Van den Bussche, J.: Data provenance for SHACL. In: Stoyanovich, J., Teubner, J., et al. (eds.) Proceedings 26th International Conference on Extending Database Technology. pp. 285–297. OpenProceedings.org (2023)
22. Eich, M., Fender, P., Moerkotte, G.: Efficient generation of query plans containing group-by, join, and groupjoin. The VLDB Journal **27**(5), 617–641 (2018)
23. Erling, O.: Implementing a SPARQL-compliant RDF triple store using a SQL-ORDBMS. `https://vos.openlinksw.com/owiki/wiki/VOS/VOSRDFWP`, retrieved 8 April 2024
24. Erling, O.: Virtuoso, a hybrid RDBMS/graph column store. IEEE Data Engineering Bulletin **35**(1), 3–8 (2012)
25. Erling, O., Mikhailov, I.: RDF support in the Virtuoso RDBMS. In: Auer, S., Bizer, C., Müller, C., Zhdanova, A. (eds.) Proceedings 1st Conference on Social Semantic Web. Lecture Notes in Informatics, vol. P-113, pp. 59–68. GI (2007)
26. Fent, P., Neumann, T.: A practical approach to groupjoin and nested aggregates. Proceedings of the VLDB Endowment **14**(11), 2383–2396 (2021)
27. Figuera, M., Rohde, P., Vidal, M.E.: Trav-SHACL: Efficiently validating networks of SHACL constraints. In: Leskovec, J., et al. (eds.) Proceedings WWW'21. pp. 3337–3348. ACM (2021)
28. Ganski, R., Wong, H.: Optimization of nested SQL queries revisited. SIGMOD Record **16**(3), 23–33 (1987)
29. Gayo, J., Prud'hommeaux, E., Boneva, I., Kontokostas, D.: Validating RDF data. Synthesis Lectures on the Semantic Web: Theory and Technology **16** (2018)
30. Ghidini, C., Hartig, O., Maleshkova, M., Svátek, V., et al. (eds.): Proceedings 18th International Semantic Web Conference, Lecture Notes in Computer Science, vol. 11778. Springer (2019)
31. Gray, J., et al.: Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. Data Mining and Knowledge Discovery **1**(1), 29–53 (1007)
32. Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C Recommendation (Mar 2013)
33. Hogan, A., Riveros, C., Rojas, C., Soto, A.: A worst-case optimal join algorithm for SPARQL. In: Ghidini et al. [30], pp. 258–275
34. Ibragimov, D., Hose, K., Bach Pedersen, T., Zimányi, E.: Processing aggregate queries in a federation of SPARQL endpoints. In: Gandon, F., Sabou, M., Sack, H., et al. (eds.) Proceedings 12th European Semantic Web Conference. Lecture Notes in Computer Science, vol. 9088, pp. 269–285. Springer (2015)
35. Kemper, A., Neumann, T.: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: Proceedings 27th International Conference on Data Engineering. pp. 195–206. IEEE Computer Society (2011)
36. Kersten, T., Leis, V., et al.: Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. Proceedings of the VLDB Endowment **11**(13), 2209–2222 (2018)
37. Knublauch, H., Kontokostas, D.: Shapes constraint language (SHACL). W3C Recommendation (Jul 2017)
38. Kohn, A., Leis, V., Neumann, T.: Tidy tuples and flying start: Fast compilation and fast execution of relational queries in Umbra. The VLDB Journal **30**(5), 883–905 (2021)
39. Labra Gayo, J.: Creating knowledge graph subsets using shape expressions. arXiv:2110.11709 (Oct 2021)

40. Leinberger, M., Seifer, P., et al.: Deciding SHACL shape containment through description logics reasoning. In: Pan, J., et al. (eds.) Proceedings 19th International Semantic Web Conference. Lecture Notes in Computer Science, vol. 12506, pp. 366–383. Springer (2020)
41. Lieber, S., Dimou, A., Verborgh, R.: Statistics about data shape use in RDF. In: Taylor, K., et al. (eds.) Proceedings of the ISWC 2020 Demos and Industry Tracks: From Novel Ideas to Industrial Practice. vol. 2721. CEUR Workshop Proceedings (2020)
42. Moerkotte, G., Neumann, T.: Accelerating queries with group-by and join by groupjoin. Proceedings of the VLDB Endowment **4**, 843–851 (2011)
43. Morfonios, K., et al.: ROLAP implementations of the data cube. ACM Computing Surveys **39**(4), 12:1–12:53 (2007)
44. Neumann, T., Weikum, G.: The RDF-3X engoine for scalable management of RDF data. The VLDB Journal **19**(1), 91–113 (2010)
45. Pareti, P., Konstantinidis, G.: A review of SHACL: From data validation to schema reasoning for RDF graphs. In: Šimkus, M., Varzinczak, I. (eds.) Reasoning Web: Declarative Artificial Intelligence. Lecture Notes in Computer Science, vol. 13100, pp. 115–144. Springer (2022)
46. Pareti, P., Konstantinidis, G., Mogavero, F.: Satisfiability and containment of recursive SHACL. Journal of Web Semantics **74**, 100721 (2022)
47. Raasveld, M., Mühleisen, H.: DuckDB: An embeddable analytical database. In: Proceedings 2019 International Conference on Management of Data. pp. 1981–1984. ACM (2019)
48. Rabbani, K., Lissandrini, M., Hose, K.: Optimizing SPARQL queries using shape statistics. In: Velegrakis, Y., Zeinalipour-Yazti, D., et al. (eds.) Proceedings 24th International Conference on Extending Database Technology. pp. 505–510. OpenProceedings.org (2021)
49. RDF 1.1 primer. W3C Working Group Note (Jun 2014)
50. Robinson, J., Voronkov, A. (eds.): Handbook of Automated Reasoning. Elsevier and MIT Press (2001)
51. Rohde, P., et al.: SHACL-ACL: Access control with SHACL. In: Pesquita, C., Skaf-Molli, H., et al. (eds.) The Semantic Web: ESWC Satellite Events. Lecture Notes in Computer Science, vol. 13998, pp. 22–26 (2023)
52. Schaffenrath, R., Proksch, D., Kopp, M., Albasini, I., Panasiuk, O., Fensel, A.: Benchmark for performance evaluation of shacl implementations in graph databases. In: V., G.B., Kliegr, T., Soylu, A., et al. (eds.) Proceedings 4th International Joint Conference on Rules and Reasoning. Lecture Notes in Computer Science, vol. 12173, pp. 82–96. Springer (2020)
53. SHACL test suite and implementation report. W3C Document (Jan 2024)
54. Shacl2sparql. `https://github.com/rdfshapes/shacl-sparql`
55. ShEx—shape expressions. `https://shex.io` (Apr 2024)
56. Stonebraker, M., et al.: C-Store: A column-oriented DBMS. In: Böhm, K., Jensen, C., et al. (eds.) Proceedings 31th International Conference on Very Large Data Bases. pp. 553–564. ACM (2005)
57. TPC benchmark H decision support standard specification revision 3.0.1. Transaction Processing Performance Council (1993–2022)
58. Trav-shacl implementation. `https://github.com/SDM-TIB/Trav-SHACL`
59. Werbrouck, J., et al.: Pattern-based access control in a decentralised collaboration environment. In: Poveda-Villalón, M., Roxin, A., et al. (eds.) Proceedings 8th Linked Data in Architecture and Construction Workshop. CEUR Workshop Proceedings, vol. 2636, pp. 118–131 (2020)