Compiling SHACL into SQL

Maxime Jakubowski[1,2] and Jan Van den Bussche[2]

[1]DBAI @ TU Wien

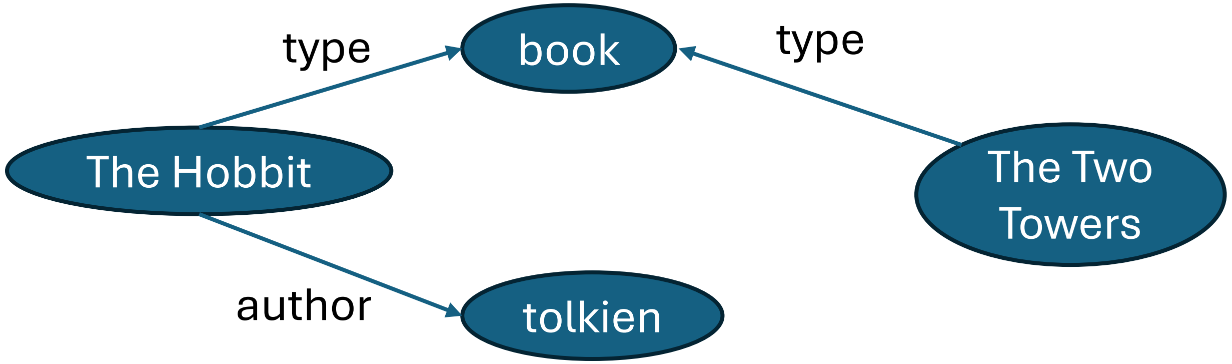
[2]DSI @ UHasselt

Shapes **C**onstraint **L**anguage (SHACL)

- *Shapes* are constraints on a graph, and consists of two parts:
  - the ***shape expression***: describes a constraint on a node
  - the ***target declaration***: descibes which nodes need to be checked
- Checking whether a shape holds in a graph is called ***validation***
- For example:
  - “has an **author**” is a shape expression
  - “every **book**” is a target declaration

```
:BookShape a sh:PropertyShape ;
    sh:path :author ;
    sh:minCount 1 .

:BookShape sh:targetClass :Book .
```



Logical SHACL

Shape Expressions

$E \coloneqq p \mid p^{-} \mid E \cdot E \mid E \cup E \mid E^{*}$ *(path expressions)*

$\phi \coloneqq \top \mid hasValue(c) \mid eq(E,p) \mid disj(E,p) \mid closed(Q)$ *(shape expressions)*

$\mid hasShape(s) \mid lessThan(E,p) \mid uniquelang(E)$

$\mid \geq_n E.\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg\phi$

- A **shape** is a triple consisting of a ***shape name***, ***expression***, and ***target***
- Target declarations can also be expressed as shapes
- For example: "sh:targetClass :Book" is $\geq_1 \text{type}.hasValue(\text{:Book})$
- We formulate our previous example in logical SHACL as a triple:

$$(\text{:BookShape}, \quad \geq_1 \text{author}, \quad \geq_1 \text{type}.hasValue(\text{:Book}))$$

Validation *is* Querying

- Shape expressions and target declarations are like node-queries:
  - shape_expr(node) retrieves all nodes satisfying a shape
  - target(node) retrieves all targeted nodes
- Retrieving all ***violating*** nodes, i.e., validation, is the difference query:

  violation(node) = target(node) – shape_expr(node)

- SHACL is a powerful language with logical constructs and we can write the violation query as a shape expression itself!
- Our example for :BookShape is then:

$$\geq_1 \text{ type}.\,hasValue(\text{:Book}) \land \neg{\geq_1} \text{ author}$$

Analytical Queries

Hypothesis SHACL validation is *analytical querying*

- Often non-monotonic queries with aggregation
  - "Qualified Value Shapes" (of the form $\geq_n p.\phi$) are group joins with count aggregation
  - Lessthan constraints are min/max aggregation queries with counting
  - Disjointness constraints and closedness are difference queries
- These are usually formulated in SQL
- Our goal:

Test this hypothesis **out of the box**

Methodology: a translation

- Validation amounts to shape evaluation, so the focus is on the translation of a single shape expression
- Given a shape expression ϕ, we define a SQL query Q_ϕ such that:

  For a given graph G, nodes satisfying ϕ in G are exactly the nodes returned by executing Q_ϕ on the relational database representation of G
- We will:
  - Define the relational representation of G
  - Utilize the *negation normal form* of ϕ to obtain an efficient translation
  - Use the SQL database DuckDB ”out-of-the-box”

Translation: the database schema

- Tries to stay close to the RDF specification
- Uses “pooling” technique (nodes get an identifier)
- Database schema:
  - **IRIs(Node: int64, Value: string)**
  - **Blanks(Node: int64)**
  - **Literals(Node: int64, Value: string, Type: string, Lang: string)**
  - **Nodes(Node: int64)**
  - **Triples(Subject: int64, Predicate: string, Object: int64)**
  - **Numerics(Node: int64, Value: double)**

Translation: Cardinality Constraints (1)

- (Qualified) min count: $\geq_n p.\phi$

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

- (Qualified) min and max count: $\geq_n p.\phi \wedge \leq_m p.\phi$

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

Translation: Cardinality Constraints (2)

- (Qualified) max count: $\leq_n p.\phi$

SELECT Node **FROM** Nodes
WHERE Node **NOT IN** ($Q_{\geq n+1\ p.\phi}$))

- (Qualified) universal: $\forall p.\phi$

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

Translation: Equality Constraint $eq(p, q)$

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

Translation: Equality Constraint $\neg eq(p, q)$

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

Translation: everything else

- Our translation covers:
  - All “logical” features:

    `sh:and`, `sh:or`, `sh:not`, `sh:xone`, `sh:disjoint`, `sh:equals`, …
  - Most of the tests on RDF Literals:

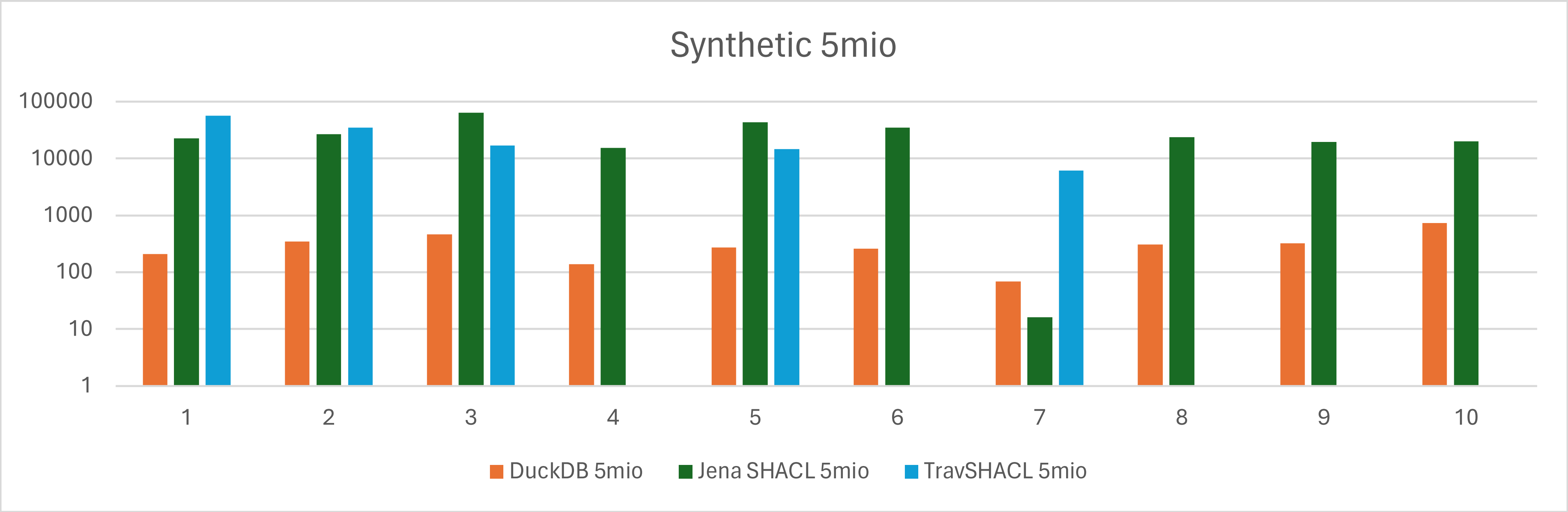
    `sh:lessThan`, `sh:hasvalue`, `sh:datatype`, `sh:pattern`, `sh:languageIn`, …
- Does not cover:
  - Full path expressions (only inverse)
  - Comparison constraints on values other than numerics
- Full description in the paper, and implemented & available online

github.com/MaximeJakubowski/shaclsql-supplementary

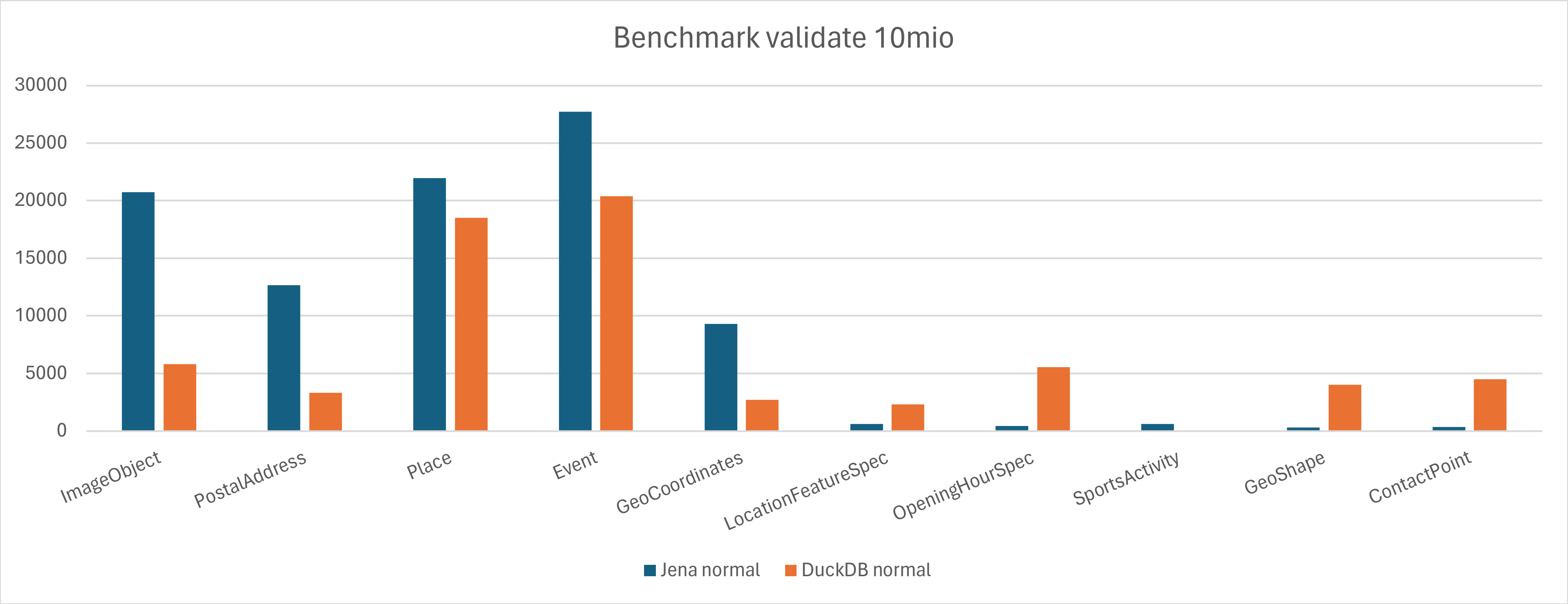
Experimental Setup

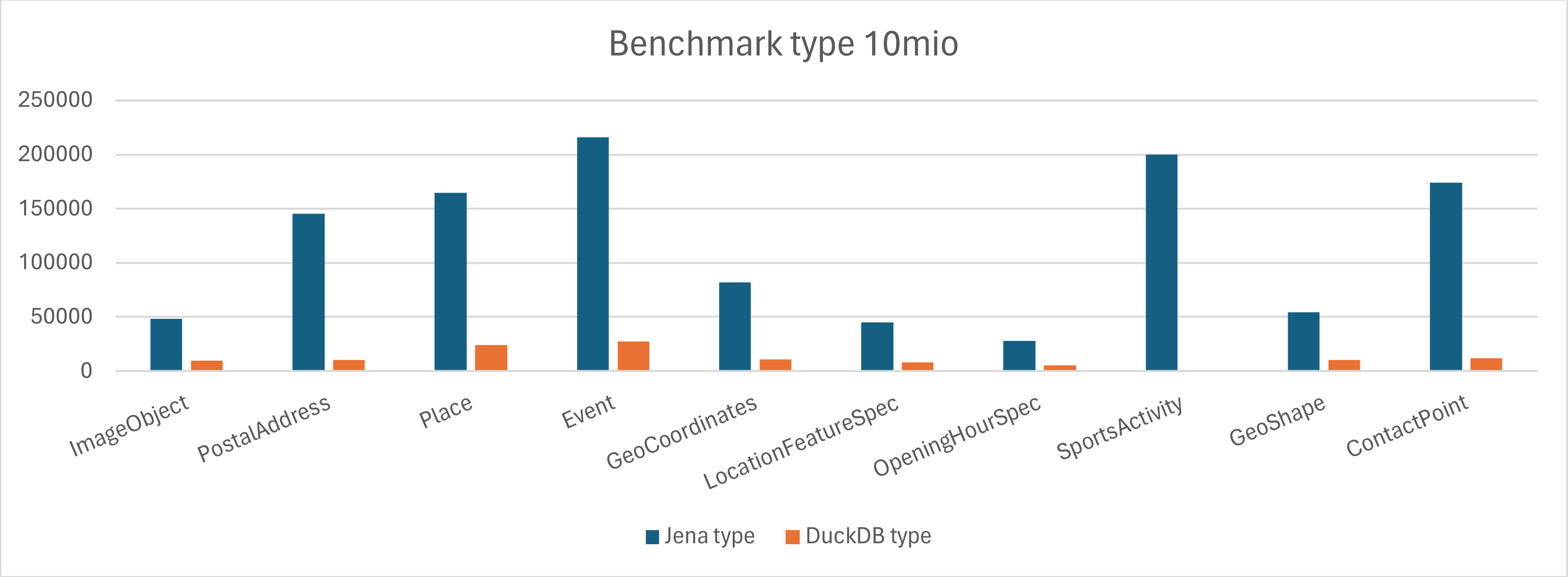
Three different experiments:

1. Synthetic shapes and data
   - covers many SHACL features
   - simple synthetic data
2. Tyrolian benchmark
   - uses real data
   - mostly conjunctions of datatype tests
3. DBLP data with custom shapes
   - covers more complex shapes
   - uses real data



Synthetic 5mio
100000
10000
1000
100
10
1
1
2
3
4
5
6
7
8
9
10
DuckDB 5mio
Jena SHACL 5mio
TravSHACL 5mio


Benchmark validate 10mio
30000
25000
20000
15000
10000
5000
0
ImageObject
PostalAddress
Place
Event
GeoCoordinates
LocationFeatureSpec
OpeningHourSpec
SportsActivity
GeoShape
ContactPoint
Jena normal
DuckDB normal


Benchmark type 10mio
250000
200000
150000
100000
50000
0
ImageObject
PostalAddress
Place
Event
GeoCoordinates
LocationFeatureSpec
OpeningHourSpec
SportsActivity
GeoShape
ContactPoint
Jena type
DuckDB type

Concuding Remarks

- Recent advances in databases are relevant for SHACL validators
- SHACL is like analytical querying
- Our approach runs best if there are many targets

- Expansion to support more complex path expressions
- Expansion to support recursion