# VIG: Data Scaling for OBDA Benchmarks

Davide Lanti, Guohui Xiao *, and Diego Calvanese
*Free University of Bozen-Bolzano*
*{dlanti,xiao,calvanese}@inf.unibz.it*

**Editor(s):** Axel-Cyrille Ngonga Ngomo, Institute for Applied Informatics, Leipzig, Germany; Irini Fundulaki, ICS-FORTH, Heraklion, Greece; Anastasia Krithara, National Center for Scientific Research "Demokritos", Athens, Greece
**Solicited review(s):** Manolis Terrovitis, IMIS, RC Athena, Athens, Greece; Panagiotis Papadakos, ICS-FORTH, Heraklion, Greece; Three Anonymous Reviewers

**Abstract.** In this paper we describe VIG, a data scaler for Ontology-Based Data Access (OBDA) benchmarks. Data scaling is a relatively recent approach, proposed in the database community, that allows for quickly scaling an input data instance to $s$ times its size, while preserving certain application-specific characteristics. The advantages of the scaling approach are that the same generator is general, in the sense that it can be re-used on different database schemas, and that users are not required to manually input the data characteristics. In the VIG system, we lift the scaling approach from the pure database level to the OBDA level, where the domain information of ontologies and mappings has to be taken into account as well. VIG is efficient and notably each tuple is generated in constant time. To evaluate VIG, we have carried out an extensive set of experiments with three datasets (BSBM, DBLP, and NPD), using two OBDA systems (Ontop and D2RQ), backed by two relational database engines (MySQL and PostgreSQL), and compared with real-world data, ad-hoc data generators, and random data generators. The encouraging results show that the data scaling performed by VIG is efficient and that the scaled data are suitable for benchmarking OBDA systems.

Keywords: Data scaling, OBDA, benchmarking

## 1. Introduction

An important research problem in Big Data is how to provide end-users with transparent access to the data, abstracting from storage details. The paradigm of Ontology-based Data Access (OBDA) [7,27] provides an answer to this problem that is very close to the spirit of the Semantic Web. In OBDA, the data stored in a relational database is presented to the end-users as a *virtual* RDF graph, whose vocabulary is provided by the classes and properties of an ontology, over which SPARQL queries can be posed. This solution is realized through *mappings* that establish a link between the ontology and the database, by specifying how to instantiate the classes and properties in the ontology by means of queries over the database.

A lot of research on optimization techniques for OBDA has been carried out recently, with the aim of making this paradigm effective in practice [11,24,4,15, 19,5,28]. In order to evaluate the effectiveness of optimizations, some benchmarks have been proposed and applied to the OBDA setting[1]. However, proper benchmarking of OBDA systems requires scalability analyses taking into account data instances of increasing volume. Such instances are often provided by generators of synthetic data. However, such generators are either complex ad-hoc implementations working for a specific schema, or require considerable manual input by the end-user. The latter problem is exacerbated in the OBDA setting, where database schemas tend to be particularly big and complex (e.g., 70 tables, some with more than 80 columns in the NPD bench-

*Corresponding Author

[1]`http://dl.kr.org/omqbench/`

mark [17]). This contributes to the slow creation of new benchmarks, and the same old benchmarks become more and more misused over a long period of time. For instance, evaluations on OBDA systems are usually performed on benchmarks originally designed for triple stores, although these two types of systems are substantially different and present different challenges [17].

Data scaling [26] is a recent approach that tries to overcome this problem by automatically tuning the generation parameters through statistics collected over an initial data instance. Hence, the same generator can be reused in different contexts, as long as an initial data instance is available. A measure of quality for the produced data is defined in terms of results for the available queries, which should be *similar* to the ones observed for real data of comparable volume.

In the context of OBDA, however, taking as the *only* parameter for generation an initial data instance does not produce data of acceptable quality, since the generated data has to comply with constraints deriving from the structure of the mappings and the ontology, which in turn derive from the application domain.

In this work we present the VIG system, a data scaler for OBDA benchmarks that addresses these issues. In VIG, the scaling approach is lifted from the instance level to the OBDA level, where the domain information of ontologies and mappings has to be taken into account as well. VIG is very efficient and suitable to generate huge amounts of data, as tuples are generated in constant time without disk accesses or need to retrieve previously generated values. Furthermore, different instances of VIG can be delegated to different machines, and parallelization can scale up to the number of columns in the schema, without communication overhead. Finally, VIG produces data in the form of csv files that can be imported easily by any relational database system.

To evaluate VIG, we have carried out an extensive set of experiments with three datasets (BSBM, DBLP, and NPD) resembling information needs in real-world use cases, using two OBDA systems (Ontop and D2RQ), backed by two relational database engines (MySQL and PostgreSQL), and compared with real-world data, ad-hoc data generators, and random data generators. In numbers, we ran in total 8042 queries over 49 database instances, among which 4 (DBLP and NPD under MySQL and PostgreSQL) are real-world data, 3 are original synthetic data (MySQL and PostgreSQL) from BSBM, and the rest are generated by VIG in different modes. The results over the three datasets are encouraging and show that the data scaling performed by VIG is efficient and that the scaled data are suitable for benchmarking OBDA systems. Moreover, the results obtained over the NPD dataset demonstrate the benefits of taking into account mappings in the generation of data.

The rest of the paper is structured as follows. In Section 2, we introduce the basic notions and notation to understand this paper. In Section 3, we define the scaling problem and discuss important measures on the produced data that define the quality of instances in a given OBDA setting. In Section 4, we discuss the VIG algorithm, and how it ensures that data conforming to the identified measures is produced. In Section 5, we provide an empirical evaluation of VIG, in terms of both resource consumption and quality of produced data, and in Section 6.1 we discuss the impact of multi-attribute foreign keys on the generation process. Sections 7 and 8 contain related work and conclusions, respectively.

This paper is a revised and significantly extended version of an article containing preliminary results that was presented at the Workshop on Benchmarking Linked Data (BLINK) 2016 [18][2].

## 2. Basic Notions and Notation

We assume that the reader has moderate knowledge of OBDA, and refer for it to the abundant literature on the subject, like [6]. Moreover, we assume familiarity with basic notions from probability calculus and statistics.

The W3C standard ontology language for OBDA is OWL 2 QL [20]. For the sake of conciseness, we consider here its mathematical underpinning *DL-Lite*$_{\mathcal{R}}$ [8]. Table 1 shows a portion of the *DL-Lite*$_{\mathcal{R}}$ ontology from the NPD benchmark, which is the foundation block of our running example.

The W3C standard query language in OBDA is SPARQL [14], with queries evaluated under the OWL 2 QL entailment regime [16]. Intuitively, under these semantics each basic graph pattern (BGP) can be seen as a single conjunctive query (CQ) without

[2]With respect to [18], apart from providing many more details on the VIG algorithm, we have also carried out more extensive experiments, by including DBLP as a third dataset (cf. Section 5.3), by using as OBDA system also D2RQ, in addition to Ontop, and by relying on PostgreSQL, in addition to MySQL, as relational database engine (cf. Section 5.1).

Table 1

Portion of the ontology for the NPD benchmark. The first three axioms (left to right) state that the classes *Exploration Wellbore* (ExpWellbore), *Shallow Wellbore* (ShWellbore), and *Suspended Wellbore* (SuspWellbore) are subclasses of the class Wellbore. The fourth axiom states that the classes ExpWellbore and ShWellbore are disjoint.

| | | |
|---|---|---|
| ExpWellbore | $\sqsubseteq$ | Wellbore |
| ShWellbore | $\sqsubseteq$ | Wellbore |
| SuspWellbore | $\sqsubseteq$ | Wellbore |
| ExpWellbore $\sqcap$ ShWellbore | $\sqsubseteq$ | $\bot$ |

existentially quantified variables. As in our examples we only refer to SPARQL queries containing exactly one BGP, we use the more concise syntax for CQs rather than the SPARQL syntax. Table 2 shows the two queries used in our running example.

The mapping component links predicates in the ontology to queries over the underlying relational database. To present our techniques, we need to introduce this component in a formal way. The standard W3C language for mappings is R2RML [10], however here we use a more concise syntax that is common in the OBDA literature. Formally, a *mapping assertion* $\mathfrak{m}$ is an expression of the form $X(\vec{f},\vec{x}) \rightsquigarrow \mathrm{conj}(\vec{y})$, consisting of a *source* part $\mathrm{conj}(\vec{y})$, which is a CQ whose output variables are $\vec{y}$, and a *target* part $X(\vec{f},\vec{x})$, which is an atom whose predicate is $X$ over terms (also called *templates*) built using function symbols $\vec{f}$ and variables $\vec{x} \subseteq \vec{y}$. We say that such a mapping $\mathfrak{m}$ *defines the predicate X*. A *basic mapping* is a mapping whose source part is a query containing exactly one atom. Table 3 shows the mappings for our running example, and provides a short description of how these mappings are used in order to create a (virtual) set of *assertions*.

Table 2

Queries for our running example.

| | |
|---|---|
| $q_1(y)$ | $\leftarrow$ Wellbore$(y)$, shWellboreForField$(x,y)$ |
| $q_2(x,n,y)$ | $\leftarrow$ Wellbore$(x)$, name$(x,n)$, complYear$(x,y)$ |

For the rest of the paper we fix an *OBDA instance* $(\mathcal{T},M,\Sigma,\mathcal{D})$, where $\mathcal{T}$ is an OWL 2 QL ontology, $\Sigma$ is a database schema with foreign and primary key dependencies, $\mathcal{M}$ is a set of mappings linking predicates in $\mathcal{T}$ to queries over $\Sigma$, and $\mathcal{D}$ is a database instance that satisfies the dependencies in $\Sigma$ and such that the assertions created from $\mathcal{D}$ via mappings $\mathcal{M}$ do not violate the disjointness axioms in $\mathcal{T}$. We denote by $\mathrm{col}(\Sigma)$ the set of all columns in $\Sigma$. Given a column $C \in \mathrm{col}(\Sigma)$, we denote by $C^{\mathcal{D}}$ the set of values for $C$ in $\mathcal{D}$. Finally, given a term $f(\vec{x})$, where $\vec{x} = (x_1,\ldots,x_p,\ldots,x_n)$, we denote the argument $x_p$ at position $p$ by $f(\vec{x})|_p$.

## 3. Data Scaling for OBDA Benchmarks: The VIG Approach

The *data scaling problem* introduced in [26] is formulated as follows:

**Definition 3.1 (Data Scaling Problem).** Given a database instance $\mathcal{D}$ for a schema $\Sigma$, and a scale factor $s$, produce a database instance $\mathcal{D}'$ for $\Sigma$ which is *similar* to $\mathcal{D}$ but $s$ times its size. ■

The notion of *similarity* is application-based. Being our goal benchmarking, we define similarity in terms of *execution times* for the evaluation of the queries in

Table 3

Mappings from the NPD benchmark.

| | | |
|---|---|---|
| ExpWellbore$(w(\texttt{id}))$ | $\rightsquigarrow$ | `exploration_wellbores(id,active,name,year)` |
| ShWellbore$(w(\texttt{id}))$ | $\rightsquigarrow$ | `shallow_wellbores(id,name,year,fid)` |
| SuspWellbore$(w(\texttt{id}))$ | $\rightsquigarrow$ | `exploration_wellbores(id,active,name,year), active='false'` |
| Field$(f(\texttt{fid}))$ | $\rightsquigarrow$ | `fields(fid,name)` |
| complYear$(w(\texttt{id}),\texttt{year})$ | $\rightsquigarrow$ | `exploration_wellbores(id,active,name,year)` |
| complYear$(w(\texttt{id}),\texttt{year})$ | $\rightsquigarrow$ | `shallow_wellbores(id,name,year,fid)` |
| name$(w(\texttt{id}),\texttt{name})$ | $\rightsquigarrow$ | `exploration_wellbores(id,active,name,year)` |
| name$(w(\texttt{id}),\texttt{name})$ | $\rightsquigarrow$ | `shallow_wellbores(id,name,year,fid)` |
| shWellboreForField$(w(\texttt{id}),f(\texttt{fid}))$ | $\rightsquigarrow$ | `shallow_wellbores(id,name,year,fid), fields(fid,fname)` |

Results from the evaluation of the queries on the source part build assertions in the ontology. For example, each tuple $(a,b,c,d)$ in a relation for `shallow_wellbores` generates an object $w(a)$ and an assertion ShWellbore$(w(a))$ in the ontology. In the R2RML mappings for the original NPD benchmark the term $w(\texttt{id})$ corresponds to the URI template `npd:wellbore/{id}`. Columns named `id` are primary keys, and the column `fid` in `shallow_wellbores` is a foreign key for the primary key `fid` of the table `fields`.

the benchmark. More precisely, consider a (real world) database instance $\mathcal{D}_0$ and a larger (real world) instance $\mathcal{D}_1$ into which $\mathcal{D}_0$ has evolved over time. Consider also an instance $\mathcal{D}'_0$ scaled from $\mathcal{D}_0$ up to a size comparable to that of $\mathcal{D}_1$. Then, the measured execution times for the evaluation of the benchmark queries over $\mathcal{D}'_0$ should be close to the measured execution times for the evaluation of the same queries over $\mathcal{D}_1$. In [26], the authors do not consider benchmark queries to be available to the generator, since their goal is broader than benchmarking over a pre-defined set of queries. In OBDA benchmarking, however, the (SQL) workload for the database can be estimated from the mapping component. Therefore, VIG includes the mappings in the analysis, so as to obtain a more realistic and OBDA-tuned generation.

Concerning the size, similarly to other approaches, VIG scales each table in $\mathcal{D}$ by a factor of $s$.

## 3.1. *Similarity Measures for OBDA and Their Rationale*

Our notion of similarity is an ideal one, but also an abstract one, as it does not provide any guideline on how the data should be generated. In this section we overview the concrete similarity measures which are used by VIG to guide the generation process. Our idea is that generating instances similar to the initial one with respect to these concrete measures can be a suitable way to produce instances that are similar also with respect to our ideal abstract similarity measure.

*Schema Dependencies.* The scaled instance $\mathcal{D}'$ should be a valid instance for $\Sigma$. VIG is, to the best of our knowledge, the only data scaler able to generate in constant time tuples that satisfy multi-attribute primary keys for *weakly-identified entities*[3]. The current algorithm of VIG supports single-attribute foreign keys, but not yet multi-attribute foreign keys. We refer to Section 6.1 for a discussion on the impact of multi-attribute foreign keys on the generation process.

*Column-based Duplicates and* `NULL` *Ratios.* These two parameters, which respectively measure the ratio of duplicates and of `NULL`s in a given column, are commonly used for the cost estimation performed by query planners in databases. By default, VIG maintains them in $\mathcal{D}'$ to preserve the cost of a number of algebra operations, such as projections or joins between columns in a key-foreign key relationship (e.g., the join from the last mapping in our running example). This default behavior, however, is not applied with *fixed-domain* columns, which are columns whose content does not depend on the size of the database instance. An example of fixed-domain column is the column `active` in the table `exploration_wellbore`, because depending on the value of `active`, the elements of `id` are partitioned into a fixed number of types, corresponding to classes in the ontology[4]. VIG analyzes the mappings to detect such cases of fixed-domain columns, and additional fixed-domain columns can be manually specified by the user. To generate values for a fixed-domain column, VIG reuses the values found in $\mathcal{D}$ so as to prevent empty answers for the SQL queries in the mappings. For instance, a value '`false`' must be generated for the column `active` in order to produce objects for the class SuspWellbore.

VIG generates values in columns according to a *uniform distribution*, that is, all values in a column have the same probability of being repeated. Replication of the distributions from $\mathcal{D}$ will be included in a future release of VIG.

*Size of Columns Clusters, and Disjointness.* Query $q_1$ from our running example returns an empty set of answers, regardless of the considered data instance. This is because $q_1$ performs a join between wellbores and fields, but the function $w$ used to build objects for the class Wellbore does not match with the function $f$ used to build objects for the class Field. Indeed, fields and wellbores are two different entities and a join between them is meaningless.

On the other hand, a standard OBDA translation of $q_2$ into SQL produces a union of CQs, in which several of these CQs contain joins between the tables `exploration_wellbores` and `shallow_wellbores`. This is possible only because the mappings for Wellbore, name, and complYear all use the *same* unary function symbol $w$ to define wellbores. Intuitively, every pair of terms over the same function symbol and appearing in the target of two distinct basic mappings identifies sets of columns for which the join operation is semantically meaningful[5]. Generating data that guar-

[3]In a relational database, a weak entity is an entity that cannot be uniquely identified by its attributes alone.

[4]The number of classes in the ontology does not depend on the size of the data instance.

[5]Therefore, between such identified sets of columns a join could occur during the evaluation of a user query.

```
function VIG($(\mathcal{T}, \mathcal{M}, \Sigma), \mathcal{D}$)
    $initAllColumns(\Sigma, \mathcal{D})$               ▷ Initialization and Satisfaction of Primary Keys
    $establishColumnsBounds(\Sigma, \mathcal{D})$
    $updateIntervalsWRTFkeys(\Sigma)$                   ▷ Satisfaction of Foreign Keys
    $\mathcal{D}' = generate()$                         ▷ Generation Phase
    return $\mathcal{D}'$
end function

function $establishColumnsBounds((\mathcal{T}, \mathcal{M}, \Sigma), \mathcal{D})$
    $fillFirstInterval(\Sigma, \mathcal{D})$            ▷ Creation of Intervals
    $CCs \leftarrow extractColumnsClusters(\Sigma, \mathcal{M})$   ▷ Columns Cluster Analysis
    for $C$ in $\Sigma$ do
        $updateIntervals(C, CCs)$                       ▷ Columns Cluster Analysis
    end for
end function
```

Fig. 1. The VIG Algorithm.

antees the correct cost for these joins is crucial in order to deliver a realistic evaluation. In our example, a join between `exploration_wellbores` and `shallow_wellbores` over the attribute `id` is empty under $\mathcal{D}$ (in fact, ExpWellbore and ShWellbore are disjoint classes, and by assumption $\mathcal{D}$ respects the disjointness constraints of the ontology). VIG is able to replicate this fact in $\mathcal{D}'$. This implies that VIG can generate data satisfying disjointness constraints declared over classes whose individuals are constructed from a unary template in a basic mapping, if $\mathcal{D}$ satisfies those constraints.

## 4. The VIG Algorithm

In this section we show how VIG realizes the measures described in the previous section. To allow the reader to concentrate on the important aspects, and considering that different database systems use different names for datatypes, we do not discuss low-level details, e.g., which domains for attributes are acceptable. We provide such details in the online resource[6] where the generator is made available.

Algorithm 1 shows a high-level view of VIG. Each comment in the code refers to a specific paragraph name in this section, where the respective code is explained and discussed. The function names in the pseudo-code recall the actual method names of the VIG implementation, which is freely available on GitHub[7] and open to contributions.

At the high-level, the VIG algorithm consists of two phases: the *analysis phase*, and the *generation phase*. In the analysis phase, the input data instance $\mathcal{D}$ is analyzed to collect statistics and create a set $\mathrm{ints}(C)$ of intervals associated to each column $C$ in the database schema. Each interval $I \in \mathrm{ints}(C)$ for a column $C$ is a structure $[\mathrm{min}, \mathrm{max}]$ keeping track of a minimum and maximum integer index. In the generation phase, a pseudo random number generator is used to randomly choose indexes from each of these intervals. Then, for each column $C$, an injective function $g_C$ is applied to transform each chosen index $i \in I$, $I \in \mathrm{ints}(C)$, into a database value according to the datatype of $C$.

From now on, let $s$ be a scale factor, and let $\mathrm{dist}(C, \mathcal{D})$ denote the number of distinct non-`NULL` values in a column $C$ in the database instance $\mathcal{D}$. Let $\mathrm{size}(T, \mathcal{D})$ denote the number of tuples occurring in the table $T$ in the database instance $\mathcal{D}$. To illustrate the algorithm, we consider a source instance $\mathcal{D}$ that contains values as shown in Figure 2, and a scaling factor $s = 2$.

### 4.1. Analysis Phase

The goal of this phase is to create intervals of indexes for each column that will be used as an input to the generation phase. These intervals are created while satisfying certain constraints over the database

[6]`https://ontop.github.io/vig/`

[7]`https://github.com/ontop/vig`

**exploration_wellbores** (abbr. `ew`)

| id | active | ... |
|---|---|---|
| 2 | true | ... |
| 4 | false | ... |
| 6 | true | ... |
| 8 | false | ... |
| 10 | false | ... |

**shallow_wellbores** (abbr. `sw`)

| id | ... |
|---|---|
| 1 | ... |
| 3 | ... |
| 5 | ... |
| 7 | ... |
| 9 | ... |

**wellbores_overview** (abbr. `wo`)

| id | ... |
|---|---|
| 1 | ... |
| . | ... |
| . | ... |
| . | ... |
| 10 | ... |

Fig. 2. Input data instance $\mathcal{D}$. Columns `id` from tables `exploration_wellbores` and `shallow_wellbores` are foreign keys of column `id` from table `wellbores_overview`.

schema, mappings, and statistical information on the source database. The analysis phase consists of the following 5 stages:

1. *initialization:* reading of the database schema and statistics over the source database;
2. *creation of intervals:* creation, starting from the statistics, of initial intervals of indexes to associate to each column;
3. *satisfaction of primary keys:* ensuring that primary key constraints are satisfied in the scaled instance by adapting the intervals boundaries;
4. *columns cluster analysis:* identification of semantically related columns through schema and mappings analysis, and adaptation of the intervals boundaries;
5. *satisfaction of foreign keys:* ensuring that foreign key constraints are satisfied in the scaled instance by adapting the intervals boundaries.

We now detail each of these stages.

*Initialization.* For each table $T$, VIG sets the number $\mathrm{size}(T,\mathcal{D}')$ of tuples to generate to $s \cdot \mathrm{size}(T,\mathcal{D})$. Then, VIG calculates the number of distinct non-`NULL` values that need to be generated for each column, given $s$ and $\mathcal{D}$. That is, for each column $C$, if $C$ is not fixed-domain then VIG sets $\mathrm{dist}(C,\mathcal{D}') := s \cdot \mathrm{dist}(C,\mathcal{D})$. Otherwise, $\mathrm{dist}(C,\mathcal{D}')$ is set to $\mathrm{dist}(C,\mathcal{D})$. To determine whether a column is fixed-domain, VIG searches in $\mathcal{M}$ for mappings of shape

$$A(f(\vec{a})) \leftsquigarrow T(\vec{b}), b_1 = c_1, \ldots, b_n = c_n$$

where $A$ is a class, $T$ is a table, $c_1,\ldots,c_n$ are constants, and $b_1,\ldots,b_n$ are columns in $\vec{b}$. For each such mapping, VIG marks the columns $b_1,\ldots,b_n$ as fixed-domain.

**Example 4.1.** For the tables in Figure 2, VIG sets $\mathrm{size}(\mathtt{ew},\mathcal{D}') = \mathrm{size}(\mathtt{sw},\mathcal{D}') = 2 \cdot 5 = 10$, and $\mathrm{size}(\mathtt{wo},\mathcal{D}') = 2 \cdot 10 = 20$. Values for statistics $\mathrm{dist}(T.\mathtt{id},\mathcal{D}')$, where $T$ is one of $\{\mathtt{ew},\mathtt{sw},\mathtt{wo}\}$, are set in the same way, because the `id` columns do not contain duplicate values. The column `ew.active` is marked as fixed-domain, because of the third mapping in Table 3. Therefore, VIG sets $\mathrm{dist}(\mathtt{ew.active}) = 2$. ■

*Creation of Intervals.* When $C$ is a numerical column, VIG initializes $\mathrm{ints}(C)$ to the interval $I_C := [\min(C,\mathcal{D}), \min(C,\mathcal{D}) + \mathrm{dist}(C,\mathcal{D}') - 1]$ of distinct values to be generated, where $\min(C,\mathcal{D})$ denotes the minimum value occurring in $C^{\mathcal{D}}$. Otherwise, if $C$ is non-numerical, $\mathrm{ints}(C)$ is initialized to the interval $I_C := [1, \mathrm{dist}(C,\mathcal{D}')]$. We recall that the elements in the intervals in $\mathrm{ints}(C)$ are transformed into values of the desired datatype by a suitable injective function in the final generation step.

**Example 4.2.** Following on our running example, VIG creates in this phase the intervals $I_{\mathtt{ew.id}} = [2,11]$, $I_{\mathtt{ew.active}} = [1,2]$, $I_{\mathtt{sw.id}} = [1,10]$, and $I_{\mathtt{wo.id}} = [1,20]$. Then, the intervals are associated to the respective columns. Namely, $\mathrm{ints}(\mathtt{ew.id}) = \{I_{\mathtt{ew.id}}\}, \ldots, \mathrm{ints}(\mathtt{wo.id}) = \{I_{\mathtt{wo.id}}\}$. ■

*Satisfaction of Primary Keys.* The generation of distinct tuples depends on the pseudo-random number generator adopted. Formally, a pseudo-random number generator is a sequence of integers $(s_i)_{i\in\mathbb{N}}$ defined through a transition function $s_k := f(s_{k-1})$. VIG adopts a particular class of pseudo-random generators, introduced in [13], and based on *multiplicative groups modulo a prime number*. Such generators are able to generate a permutation of the indexes in an interval. In formal terms, let $n$ be the number of distinct values to generate, and let $g$ be a generator for the multiplicative group modulo a prime number $p$, with $p > n$. Consider the sequence $S := \langle g^i \ mod \ p \mid i = 1,\ldots,p \text{ and } (g^i \ mod \ p) \leq n\rangle$. Then $S$ is a *permutation* of values in the interval $[1,\ldots,n]$.

We now explain how such number generator can be used to generate tuples of values satisfying the primary key constraints in the schema. Let $K = \{C_1, \dots, C_n\}$ be the primary key of a table $T$. In order to ensure that values generated for each column through our pseudo-random generator do not lead to tuples with duplicate values for the key $K$, it suffices that the least common multiple $\mathrm{lcm}(\mathrm{dist}(C_1, \mathcal{D}'), \dots, \mathrm{dist}(C_n, \mathcal{D}'))$ of the number of distinct values to generate in each column is greater than the number $\mathrm{tuples}(T, \mathcal{D}')$ of tuples to generate for the table $T$. While this condition is not satisfied, VIG increases by 1 $\mathrm{dist}(C_i, \mathcal{D}')$ for a suitably chosen column $C_i$ in $K$. Observe that the only side effect of this is a small deviation on the number of distinct values to generate for a column. Once the condition holds, data can be generated independently for each column without risk of generating duplicate tuples for $K$.

*Columns Cluster Analysis.* In this phase, VIG analyzes $\mathcal{M}$ in order to identify columns that could be joined in a translation to SQL, and groups them together into *pre-clusters*. Formally, let $X_1(\vec{f_1}, \vec{x_1}), \dots, X_m(\vec{f_m}, \vec{x_m})$ be the atoms defined by basic mappings in $\mathcal{M}$, where variables correspond to qualified column names[8]. Consider the set $\mathcal{F} = \bigcup_{i=1\dots m} \{f(\vec{x}) \mid f(\vec{x}) \text{ is a term in } X_i(\vec{f_i}, \vec{x_i})\}$ of all the terms occurring in such atoms. For each function $f$ and valid position $p$ in $f$, we define a *pre-cluster* $\mathfrak{pc}_{f|_p}$ as the set of columns $\mathfrak{pc}_{f|_p} = \{f(\vec{x})|_p \mid f(\vec{x}) \in \mathcal{F}\}$.

**Example 4.3.** For our running example, we have

$$\mathcal{F} = \{w(\texttt{ew.id}), w(\texttt{sw.id}), f(\texttt{fields.fid})\}.$$

There are two pre-clusters, namely $\mathfrak{pc}_{w|_1} = \{\texttt{ew.id}, \texttt{sw.id}\}$ and $\mathfrak{pc}_{f|_1} = \{\texttt{fields.fid}\}$. ■

VIG evaluates on $\mathcal{D}$ all combinations of joins between columns in a pre-cluster $\mathfrak{pc}$, and produces values in $\mathcal{D}'$ so that the selectivities for these joins are maintained. In order to do so, the intervals for the columns in $\mathfrak{pc}$ are modified. This modification must be propagated to all the columns related via a foreign key relationship to some column in $\mathfrak{pc}$. In particular, the modification might propagate to columns belonging to different pre-clusters, inducing a clash. VIG groups together such pre-clusters in order to avoid this issue. Formally, let $\mathcal{PC}$ denote the set of pre-clusters for $\mathcal{M}$. For a pre-cluster $\mathfrak{pc} \in \mathcal{PC}$, let $\mathcal{C}(\mathfrak{pc}) = \{D \in \mathrm{col}(\Sigma) \mid \text{there is a } C \in \mathfrak{pc} : D \overset{*}{\leftrightarrow} C\}$, where $\overset{*}{\leftrightarrow}$ is the reflexive, symmetric, and transitive closure of the single column foreign key relation between pairs of columns[9]. Two pre-clusters $\mathfrak{pc}_1, \mathfrak{pc}_2 \in \mathcal{PC}$ are in *merge relation*, denoted as $\mathfrak{pc}_1 \leftrightsquigarrow \mathfrak{pc}_2$, iff $\mathcal{C}(\mathfrak{pc}_1) \cap \mathcal{C}(\mathfrak{pc}_2) \neq \emptyset$. Given a pre-cluster $\mathfrak{pc}$, the set $\{c \in \mathrm{col}(\Sigma) \mid c \in \mathfrak{pc}' \text{ for some } \mathfrak{pc}' \text{ with } \mathfrak{pc}' \overset{*}{\leftrightsquigarrow} \mathfrak{pc}\}$ of columns is called a *columns cluster*, where $\overset{*}{\leftrightsquigarrow}$ is the transitive closure of $\leftrightsquigarrow$. Columns clusters group together those pre-clusters for which columns cannot be generated independently.

**Example 4.4.** In our example we have that $\mathfrak{pc}_{w|_1} \overset{*}{\not\leftrightsquigarrow} \mathfrak{pc}_{f|_1}$. In fact, $\mathcal{C}(\mathfrak{pc}_{w|_1}) = \mathfrak{pc}_{w|_1} \cup \{\texttt{wo.id}\}$ and $\mathcal{C}(\mathfrak{pc}_{f|_1}) = \mathfrak{pc}_{f|_1}$, hence $\mathcal{C}(\mathfrak{pc}_{w|_1}) \cap \mathcal{C}(\mathfrak{pc}_{f|_1}) = \emptyset$. Therefore, the pre-clusters $\mathfrak{pc}_{w|_1}$ and $\mathfrak{pc}_{f|_1}$ are also columns clusters. ■

After identifying columns clusters, VIG analyzes the number of shared elements between the columns in a cluster, and creates new intervals accordingly. Formally, consider a columns cluster $\mathfrak{cc}$. Let $H \subseteq \mathfrak{cc}$ be a set of columns, and let $\mathcal{K}_H := \{K \mid H \subset K \subseteq \mathfrak{cc}\}$ be the set of strict super-sets of $H$. For each such $H$, VIG creates an interval $I_H$ of indexes such that $|I_H| := s \cdot |\bigcap_{C \in H} C^{\mathcal{D}} \setminus (\bigcup_{K \in \mathcal{K}_H} \bigcap_{C \in K} C^{\mathcal{D}})|$, and adds $I_H$ to $\mathrm{ints}(C)$ for all $C \in H$. Boundaries for all intervals $I_H$ are set in a way that they do not overlap.

**Example 4.5.** Consider the columns cluster $\mathfrak{pc}_{w|_1}$. There are three non-empty subsets of $\mathfrak{pc}_{w|_1}$, namely $E = \{\texttt{ew.id}\}$, $S = \{\texttt{sw.id}\}$, and $ES = \{\texttt{ew.id}, \texttt{sw.id}\}$. Accordingly, we identify the sets $\mathcal{K}_E = \mathcal{K}_S = \{ES\}$ and $\mathcal{K}_{ES} = \emptyset$.

Thus, VIG needs to create three disjoint intervals $I_E$, $I_S$, and $I_{ES}$ such that

- $|I_E| = 2 \cdot |\texttt{ew.id}^{\mathcal{D}} \setminus \emptyset| = 2 \cdot 5 = 10$,
- $|I_S| = 2 \cdot |\texttt{sw.id}^{\mathcal{D}} \setminus \emptyset| = 2 \cdot 5 = 10$,
- $|I_{ES}| = 2 \cdot |(\texttt{ew.id}^{\mathcal{D}} \cap \texttt{sw.id}^{\mathcal{D}}) \setminus \emptyset| = 0$.

Without loss of generality, we assume that the intervals generated by VIG satisfying the constraints above are $I_E = [2, 11]$ and $I_S = [12, 21]$. These intervals are assigned to columns `ew.id` and `sw.id`, respectively. Intervals assigned in the initialization phase to the same columns are deleted. ■

[8] A qualified column name is a string of the form `T.C`, where `T` is a table name and `C` a column name.

[9] We recall that VIG does not allow for multi-attribute foreign keys.

*Satisfaction of Foreign Keys.* At this point, foreign key columns $D$ for which there is no columns cluster $\mathfrak{cc}$ such that $D \in \mathcal{C}(\mathfrak{cc})$, have a single interval whose boundaries have to be aligned to the (single) interval of the parent. Foreign keys relating pairs of columns in a cluster, instead, are already satisfied by construction of the intervals in the columns cluster. More work, instead, is necessary for columns belonging to $\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}$, for some columns cluster $\mathfrak{cc}$. These are columns that will never appear in a join condition, given the input mappings, but that still are directly or indirectly related through a foreign key. We need to assign intervals for such columns such that the foreign keys are satisfied, in addition to all other constraints relative to duplicates ratio, primary keys, and already found intervals in columns clusters.

VIG encodes the problem of finding the right intervals for these columns into a *constraint satisfaction problem (CSP)* [1], which can be solved by any off-the-shelf constraint solver. In VIG, we use Choco [21].

Table 4 shows the CSP program that encodes the problem of establishing the intervals for the columns in $\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}$. The first constraint declares a pair of variables for each column $C \in \mathcal{C}(\mathfrak{cc})$ and interval $I$ in the set $S$ of intervals for the columns cluster $\mathfrak{cc}$. The variable $X_{\langle C, I \rangle}$, encodes the lower bound for $I$ in column $C$, and the variable $Y_{\langle C, I \rangle}$ the upper bound. The second constraint fixes the values of variables $X$ and $Y$ for all those columns belonging to the columns cluster $\mathcal{C}(\mathfrak{cc})$, as their intervals were found during the columns cluster analysis and do not need to be recomputed. The third constraint states that intervals that do not belong to a column should be set to empty for that column, and the fourth constraint states that lower-bounds should be at most as large as upper bounds. The fifth constraint states that if two columns are in a foreign-key relation $C_1 \subseteq C_2$ and belong to $\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}$, then any lower bound $X_{\langle C_1, I \rangle}$ (resp., upper bound $Y_{\langle C_1, I \rangle}$) must be greater (resp., smaller) or equal than the lower bound $X_{\langle C_2, I \rangle}$ (resp., upper bound $Y_{\langle C_2, I \rangle}$). Intuitively, this constraint will lead to the creation of an interval $I_{C_1, C_2}$ shared between $C_1$ and $C_2$. The last constraint states that the sum of the differences between upper and lower bounds variables for a column $C$ should be equal to the number of distinct elements that we want to generate for $C$.

Table 4

CSP Program for foreign keys satisfaction.

Create Program Variables:
$\forall I \in S.\, \forall C \in \mathcal{C}(\mathfrak{cc}).\, X_{\langle C, I \rangle}, Y_{\langle C, I \rangle} \in [I.min, I.max]$

Set Boundaries for Known Intervals:
$\forall I \in S.\, \forall C \in \mathcal{C}(\mathfrak{cc}).\, I \in \mathrm{ints}(C) \Rightarrow X_{\langle C, I \rangle} = I.min, Y_{\langle C, I \rangle} = I.max$

Set Boundaries for Known Empty Intervals:
$\forall I \in S.\, \forall C \in \mathfrak{cc}.\, I \notin \mathrm{ints}(C) \Rightarrow X_{\langle C, I \rangle} = Y_{\langle C, I \rangle}$

The X's should be at most as large as the Y's:
$\forall I \in S.\, \forall C \in \mathcal{C}(\mathfrak{cc}).\, X_{\langle C, I \rangle} \leq Y_{\langle C, I \rangle}$

Foreign Keys (denoted by $\subseteq$):
$\forall I \in S.\, \forall C_1 \in (\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}).\, \forall C_1 \subseteq C_2.\, X_{\langle C_1, I \rangle} \geq X_{\langle C_2, I \rangle}$
$\forall I \in S.\, \forall C_1 \in (\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}).\, \forall C_1 \subseteq C_2.\, Y_{\langle C_1, I \rangle} \leq Y_{\langle C_2, I \rangle}$
$\forall I \in S.\, \forall C_1 \in (\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}).\, \forall C_2 \subseteq C_1.\, X_{\langle C_2, I \rangle} \geq X_{\langle C_1, I \rangle}$
$\forall I \in S.\, \forall C_1 \in (\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}).\, \forall C_2 \subseteq C_1.\, Y_{\langle C_2, I \rangle} \leq Y_{\langle C_1, I \rangle}$

Width of the Intervals:
$\forall C \in (\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}).\, \sum_{I \in S} Y_{\langle C, I \rangle} - X_{\langle C, I \rangle} = |C|$

In this program, $S$ is the set of intervals for the columns in the columns cluster $\mathfrak{cc}$, plus one extra disjoint interval. Each interval $I$ in a column $C$ is encoded as a pair of variables $X_{\langle C, I \rangle}, Y_{\langle C, I \rangle}$, keeping respectively the lower and upper limit for the interval.

**Example 4.6.** At this point, the intervals found by VIG (w.r.t. the portion of $\mathcal{D}$ we are considering) are:

- $I_{\texttt{wo.id}} = [1, 20]$ and $I_{\texttt{ew.active}} = [1, 2]$, found in the initialization phase;
- $I_E = [2, 11]$ and $I_S = [12, 21]$, found during the columns cluster analysis.

Observe that these intervals violate the foreign key $\texttt{sw.id} \subseteq \texttt{wo.id}$, because the index 21 belongs to $I_{\{\texttt{sw.id}\}}$ but not to $I_{\texttt{wo.id}}$. Moreover, $\texttt{wo.id} \in \mathcal{C}(\mathfrak{pc}_{w|_1}) \setminus \mathfrak{pc}_{w|_1}$. Therefore, VIG encodes the problem of finding the right boundaries for $I_{\texttt{wo.id}}$ into the CSP in Table 5. Any solution for this CSP program sets $X_{\langle \texttt{ew.id}, I_{\texttt{ew.id}} \rangle} = 2$, $Y_{\langle \texttt{ew.id}, I_{\texttt{ew.id}} \rangle} = 11$, $X_{\langle \texttt{sw.id}, I_{\texttt{sw.id}} \rangle} = 12$, and $Y_{\langle \texttt{sw.id}, I_{\texttt{sw.id}} \rangle} = 21$. The last four constraint imply also that, for any solution, $X_{\langle C, I_{\text{aux}} \rangle} = Y_{\langle C, I_{\text{aux}} \rangle}$, for any column $C$. A solution for the program is:

$$\begin{array}{ll}
X_{\langle \texttt{ew.id}, I_{\texttt{ew.id}} \rangle} = 2 & Y_{\langle \texttt{ew.id}, I_{\texttt{ew.id}} \rangle} = 11 \\
X_{\langle \texttt{sw.id}, I_{\texttt{sw.id}} \rangle} = 12 & Y_{\langle \texttt{sw.id}, I_{\texttt{sw.id}} \rangle} = 21 \\
X_{\langle \texttt{wo.id}, I_{\texttt{ew.id}} \rangle} = 2 & Y_{\langle \texttt{wo.id}, I_{\texttt{ew.id}} \rangle} = 11 \\
X_{\langle \texttt{wo.id}, I_{\texttt{sw.id}} \rangle} = 12 & Y_{\langle \texttt{wo.id}, I_{\texttt{ew.id}} \rangle} = 21
\end{array}$$

and arbitrary values for the other variables so that $X_{\langle C, I \rangle} = Y_{\langle C, I \rangle}$.

From this solution, VIG creates two new intervals $I_{\{\texttt{wo.id}, \texttt{ew.id}\}} = [2, 11]$ and $I_{\{\texttt{wo.id}, \texttt{sw.id}\}} = [12, 21]$ and sets them as intervals for column $\texttt{wo.id}$. ■

### 4.1.1. Complexity of the Analysis Phase

We analyze the complexity of the 5 stages of the analysis phase.

1. *Initialization:* The gathering of each statistic is performed by issuing $|\operatorname{col}(\Sigma)|$ SQL queries to the

Table 5

CSP instance for the running example.

Create Program Variables:

$$X_{\langle \texttt{ew.id}, I_{\texttt{ew.id}} \rangle}, Y_{\langle \texttt{ew.id}, I_{\texttt{ew.id}} \rangle}, X_{\langle \texttt{sw.id}, I_{\texttt{ew.id}} \rangle}, Y_{\langle \texttt{sw.id}, I_{\texttt{ew.id}} \rangle}, X_{\langle \texttt{wo.id}, I_{\texttt{ew.id}} \rangle}, Y_{\langle \texttt{wo.id}, I_{\texttt{ew.id}} \rangle} \in [2, 11]$$
$$X_{\langle \texttt{ew.id}, I_{\texttt{sw.id}} \rangle}, Y_{\langle \texttt{ew.id}, I_{\texttt{sw.id}} \rangle}, X_{\langle \texttt{sw.id}, I_{\texttt{sw.id}} \rangle}, Y_{\langle \texttt{sw.id}, I_{\texttt{sw.id}} \rangle}, X_{\langle \texttt{wo.id}, I_{\texttt{sw.id}} \rangle}, Y_{\langle \texttt{wo.id}, I_{\texttt{sw.id}} \rangle} \in [12, 21]$$
$$X_{\langle \texttt{ew.id}, I_{\texttt{aux}} \rangle}, Y_{\langle \texttt{ew.id}, I_{\texttt{aux}} \rangle}, X_{\langle \texttt{sw.id}, I_{\texttt{aux}} \rangle}, Y_{\langle \texttt{sw.id}, I_{\texttt{aux}} \rangle}, X_{\langle \texttt{wo.id}, I_{\texttt{aux}} \rangle}, Y_{\langle \texttt{wo.id}, I_{\texttt{aux}} \rangle} \in [22, 41]$$

Set Boundaries for Known Intervals:

$$X_{\langle \texttt{ew.id}, I_{\texttt{ew.id}} \rangle} = 2 \qquad Y_{\langle \texttt{ew.id}, I_{\texttt{ew.id}} \rangle} = 11$$
$$X_{\langle \texttt{sw.id}, I_{\texttt{sw.id}} \rangle} = 12 \qquad Y_{\langle \texttt{sw.id}, I_{\texttt{sw.id}} \rangle} = 21$$

Set Boundaries for Known Empty Intervals:

$$X_{\langle \texttt{ew.id}, I_{\texttt{sw.id}} \rangle} = Y_{\langle \texttt{ew.id}, I_{\texttt{sw.id}} \rangle} \qquad X_{\langle \texttt{ew.id}, I_{\texttt{aux}} \rangle} = Y_{\langle \texttt{ew.id}, I_{\texttt{aux}} \rangle}$$
$$X_{\langle \texttt{sw.id}, I_{\texttt{ew.id}} \rangle} = Y_{\langle \texttt{sw.id}, I_{\texttt{ew.id}} \rangle} \qquad X_{\langle \texttt{sw.id}, I_{\texttt{aux}} \rangle} = Y_{\langle \texttt{sw.id}, I_{\texttt{aux}} \rangle}$$

The X's should be at most as large as the Y's:

$$X_{\langle \texttt{ew.id}, I_{\texttt{ew.id}} \rangle} \leq Y_{\langle \texttt{ew.id}, I_{\texttt{ew.id}} \rangle} \qquad X_{\langle \texttt{sw.id}, I_{\texttt{ew.id}} \rangle} \leq Y_{\langle \texttt{sw.id}, I_{\texttt{ew.id}} \rangle} \qquad X_{\langle \texttt{wo.id}, I_{\texttt{ew.id}} \rangle} \leq Y_{\langle \texttt{wo.id}, I_{\texttt{ew.id}} \rangle}$$
$$X_{\langle \texttt{ew.id}, I_{\texttt{sw.id}} \rangle} \leq Y_{\langle \texttt{ew.id}, I_{\texttt{sw.id}} \rangle} \qquad X_{\langle \texttt{sw.id}, I_{\texttt{sw.id}} \rangle} \leq Y_{\langle \texttt{sw.id}, I_{\texttt{sw.id}} \rangle} \qquad X_{\langle \texttt{wo.id}, I_{\texttt{sw.id}} \rangle} \leq Y_{\langle \texttt{wo.id}, I_{\texttt{sw.id}} \rangle}$$
$$X_{\langle \texttt{ew.id}, I_{\texttt{aux}} \rangle} \leq Y_{\langle \texttt{ew.id}, I_{\texttt{aux}} \rangle} \qquad X_{\langle \texttt{sw.id}, I_{\texttt{aux}} \rangle} \leq Y_{\langle \texttt{sw.id}, I_{\texttt{aux}} \rangle} \qquad X_{\langle \texttt{wo.id}, I_{\texttt{aux}} \rangle} \leq Y_{\langle \texttt{wo.id}, I_{\texttt{aux}} \rangle}$$

Foreign Keys:

$$X_{\langle \texttt{ew.id}, I_{\texttt{ew.id}} \rangle} \geq X_{\langle \texttt{wo.id}, I_{\texttt{ew.id}} \rangle} \quad \cdots \quad X_{\langle \texttt{sw.id}, I_{\texttt{sw.id}} \rangle} \geq X_{\langle \texttt{wo.id}, I_{\texttt{sw.id}} \rangle}$$
$$Y_{\langle \texttt{ew.id}, I_{\texttt{ew.id}} \rangle} \leq Y_{\langle \texttt{wo.id}, I_{\texttt{ew.id}} \rangle} \quad \cdots \quad Y_{\langle \texttt{sw.id}, I_{\texttt{sw.id}} \rangle} \leq Y_{\langle \texttt{wo.id}, I_{\texttt{sw.id}} \rangle}$$

Width of the Intervals:

$$Y_{\langle \texttt{wo.id}, I_{\texttt{ew.id}} \rangle} - X_{\langle \texttt{wo.id}, I_{\texttt{ew.id}} \rangle} + Y_{\langle \texttt{wo.id}, I_{\texttt{sw.id}} \rangle} - X_{\langle \texttt{wo.id}, I_{\texttt{sw.id}} \rangle} + Y_{\langle \texttt{wo.id}, I_{\texttt{aux}} \rangle} - X_{\langle \texttt{wo.id}, I_{\texttt{aux}} \rangle} = 20$$

source data instance, and evaluating each of them is AC0 in data complexity.

2. *Creation of intervals:* We need to create $|\operatorname{col}(\Sigma)|$ intervals, and the creation of each of these is done in constant time using the gathered statistics.
3. *Satisfaction of primary keys:* This requires a repeated computation of the least common multiple of numbers of distinct values for the columns participating in a primary key. The number of such repetitions is expected to be very small.
4. *Columns cluster analysis:* The complexity of this stage depends on the number and size of the columns clusters. The number of (non-singleton) columns clusters can range from a single columns cluster containing all the columns in $\operatorname{col}(\Sigma)$ to $|\operatorname{col}(\Sigma)|/2$ columns clusters, each of size exactly 2. The complexity of the analysis for a columns cluster $\mathfrak{cc}$ grows with the number of columns in $\mathcal{C}(\mathfrak{cc})$. Let $\mathfrak{cc}$ be the columns cluster with the largest number $n$ of columns in $\mathcal{C}(\mathfrak{cc})$. In the worst case, VIG issues to the database $O(2^n)$ queries, so as to compute for each $H \subseteq \mathcal{C}(\mathfrak{cc})$ the cardinality $|I_H|$ of the interval $I_H$ to be created. Hence, in the worst case where all columns of $\Sigma$ are in $\mathcal{C}(\mathfrak{cc})$, $O(2^{|\operatorname{col}(\Sigma)|})$ intervals will be created in this stage. One has to notice, however, that a columns cluster $\mathfrak{cc}$ groups together in $\mathcal{C}(\mathfrak{cc})$ those columns in the database that are *semantically related* (e.g., columns in a key-foreign key relationship). In practice we expect such groups to contain only a few columns[10].
5. *Satisfaction of foreign keys:* From Table 4, it is immediate to see that for each columns cluster $\mathfrak{cc}$, the CSP can contain a number of variables that is at most $O(N \cdot n)$, and a number of rules that is at most $O(\mathit{fk} \cdot N \cdot n)$, where $\mathit{fk}$ is the number of foreign keys, and $N$ is the size of the set $S$ containing all the intervals in $\mathcal{C}(\mathfrak{cc})$ (which are at most $2^n$). We observe that both the number of rules and the number of variables do not depend on the size of the initial or scaled data. Hence, the complexity of the CSP program does not depend on the data, but only on the complexity of the schema and mappings.

## 4.2. Generation Phase

At this point, each column in $\operatorname{col}(\Sigma)$ is associated to a set of intervals. The elements in the intervals are associated to values in the column datatype, and to values from $C^{\mathcal{D}}$ in case $C$ is fixed-domain. VIG uses the

[10]For NPD, we have 17 columns clusters mostly of size 2 or 3, and there is a single largest group of size 11.

**exploration_wellbores** (abbr. `ew`)

| id | active | ... |
|---|---|---|
| **2** | true | ... |
| **3** | false | ... |
| **...** | false | ... |
| **10** | true | ... |
| **11** | false | ... |

**shallow_wellbores** (abbr. `sw`)

| id | ... |
|---|---|
| **12** | ... |
| **13** | ... |
| **...** | ... |
| **20** | ... |
| **21** | ... |

**wellbores_overview** (abbr. `wo`)

| id | ... |
|---|---|
| **2** | ... |
| **...** | ... |
| **11** | ... |
| **12** | ... |
| **...** | ... |
| **21** | ... |

Fig. 3. Scaled instance $\mathcal{D}'$.

pseudo-random number generator to randomly pick elements from the intervals that are then transformed into database values. `NULL` values are generated according to the detected `NULL` ratio. Observe that the generation of a value in a column takes constant time and can happen independently for each column, thanks to the previous phases in which intervals were calculated.

**Example 4.7.** At this stage, VIG has available all the information necessary to proceed with the generation. In our running example, such information is:

– *Association of columns to intervals.* The columns in the considered tables are associated to intervals in the following way:

$$\begin{aligned}\text{ints}(\texttt{ew.id}) &= \{[2,11]\}\\ \text{ints}(\texttt{sw.id}) &= \{[12,21]\}\\ \text{ints}(\texttt{wo.id}) &= \{[2,11],[12,21]\}\\ \text{ints}(\texttt{ew.active}) &= \{[1,2]\}\end{aligned}$$

– *Number of tuples to generate for each table.* From Example 4.1, we know that $\text{size}(\texttt{ew},\mathcal{D}') = 10$, $\text{size}(\texttt{sw},\mathcal{D}') = 10$, and $\text{size}(\texttt{wo},\mathcal{D}') = 20$.

– *Association of indexes to database values.* Without loss of generality, we assume that the injective function used by VIG to associate elements in the intervals to database values is the identity function for all non fixed-domain columns. For the column `ew.active`, we use the function $g : \{1,2\} \to \{\texttt{'true'},\ \texttt{'false'}\}$ such that $g(1) = \texttt{'true'}$ and $g(2) = \texttt{'false'}$.

Figure 3 shows the generated data instance $\mathcal{D}'$. Observe that $D'$ satisfies all the constraints discussed in the previous paragraphs. For clarity, the generated tuples are sorted on the primary key, however in a real execution the values would be randomly generated by means of the multiplicative group modulo a prime number. ■

*4.2.1. Complexity of the Generation Phase*

In VIG, the complexity of the generation phase depends only on the complexity of the underlying pseudo-random number generator. The pseudo-random number generator underlying VIG requires constant time and space to produce each value [13]. Hence, the overall running time is linear in the size of the generated data.

## 5. VIG in Action

The data generation techniques presented in the previous sections have been implemented in the VIG system, which is available on GitHub[11] as a Java Maven project, and comes with documentation in form of wiki pages. Initially, VIG was implement as part of the NPD benchmark [17]. Now it has become a mature implementation delivered since two years. The system is licensed under Apache 2.0, and maintained at the Free University of Bozen-Bolzano.

In this section we present an extensive evaluation of VIG. The goal of the evaluation is to demonstrate the quality of the data scaled by VIG, by comparing the scaled data with the original data. We define quality in terms of how well the observed performance for query answering over the scaled data approximates the observed performance for query answering over the original data. To perform the comparisons, we use two OBDA systems (Ontop [5] and D2RQ[12]), backed by different relational engines (MySQL and PostgreSQL). Moreover, we compare VIG scaled data to randomly generated data.

In Section 5.1 we describe the design of our experiments, in Sections 5.2 to 5.4 we present three experiments of VIG using the BSBM, DBLP, and NPD datasets respectively, and finally in Section 6 we discuss the results of the experiments.

[11]`https://github.com/ontop/vig`
[12]`http://d2rq.org/`

Table 6
Overview of the data and design of the experiments.

| Data Set | Data characteristic | | | | Experiment design | | | |
|---|---|---|---|---|---|---|---|---|
| | Generator | Real-world data | Mappings | Ontology | Data generation | Query evaluation | Predicate growth | Use of domain info |
| BSBM | ✓ | - | ✓ | - | ✓ | ✓ | ✓ | - |
| DBLP | - | ✓ | ✓ | - | - | ✓ | ✓ | - |
| NPD | - | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ |

### 5.1. Experiment Design

For each dataset in our experiments, we consider the following characteristics:

- *generator*: whether the dataset includes a data generator;
- *real-world data*: whether the dataset includes real-world data for evaluation;
- *mappings*: whether the dataset includes mappings;
- *ontology*: whether the dataset includes an ontology.

Then we design the following experiments according to the characteristics of each dataset:

- *Data generation*: it compares the performance of data generation between the generator provided by the dataset and VIG.
- *Query evaluation*: it evaluates the test queries over different scaled data generated by VIG, the native data generator (if available), and a random data generator.
- *Predicate growth*: it evaluates how scaling affects the growth of classes and properties.
- *Use of domain info*: it evaluates the impact on the quality of the produced data of using, for the data generation, the domain information provided by the mappings and the ontology.

For the evaluation, we provide three experiments. In the first experiment we use the BSBM benchmark [2] to compare the synthetic data generated by VIG with the one generated by the native ad-hoc BSBM data generator. In the second experiment we use the real-world DBLP dataset [12] about authors and publications, and compare it to the data generated by VIG. The last experiment, which uses the NPD benchmark [17], focuses on testing the impact of the mappings analysis on the quality of the scaled data.

The characteristics of these datasets and the designed experiments are summarized in Table 6, in which "✓" means yes and "-" means no or impossible. In particular, we only evaluate data generation in BSBM because the other datasets do not include a generator, and we only evaluate the impact of an ontology in NPD as only NPD includes an expressive ontology.

We do not evaluate here the ability of VIG to generate data in parallel. We just observe that VIG generates the data for the various columns independently of each other, and as an immediate consequence the algorithm can be parallelized up to the number of columns. However, the actual impact of such parallelization on the performance largely depends on the adopted parallel system architecture (e.g., how concurrent write accesses to the storage layer by multiple threads/processors are handled). Experiments on this would test the parallel system architecture rather than the algorithm, and hence go beyond the scope of this paper.

*Global Setting for All Experiments.* For the DBLP and BSBM experiments we used two different OBDA systems, namely D2RQ v0.8.1 and Ontop v1.18.0 [5]. We used PostgreSQL v9.6.2 and MySQL v5.7.17 as underlying database engines. The choice to test on multiple systems and RDBMSs was made in order to ensure that the observed results do not depend on the actual implementation details of the considered system/RDBMS. The hardware used is a PC desktop with 4 Intel(R) Xeon(R) E5-2680 0 @ 2.70 GHz processors and 8 GB of RAM. The OS is Ubuntu 16.04 LTS.

For the NPD experiment, we changed our hardware, and used an HP Proliant server with 2 Intel Xeon X5690 Processors (each with 12 logical cores at 3.47 GHz), 106 GB of RAM and five 1TB 15K RPM HDs. We underline that the change of hardware does not affect the validity of the experiments, as we never compare results run on different machines.

The experiments were run in the OBDA-Mixer system[13], which is an automated testing platform, shipped with the NPD benchmark, that instantiates a set of template-queries with values from the data instance so as to produce SPARQL queries to be tested against an

[13]`https://github.com/ontop/obda-mixer`

OBDA query answering system. The set of queries resulting from this instantiation is called *query mix* (or, simply, *mix*). The rationale behind query mixes is to provide different variations of the test queries so as to reduce the impact of caching, in the OBDA system or in the database engine, on the measured execution times. Each mix was run 4 times: 1 warm-up run and 3 test runs. Each query was executed with a timeout of 20 minutes. We treated failed executions as timeouts.

All the material used for the experiments is made available online[14].

## 5.2. *BSBM Experiment*

The BSBM benchmark is built around an e-commerce use case in which different vendors offer products that can be reviewed by customers. It comes with a set of template-queries, an ontology, mappings, and a native ad-hoc data generator (NTV) that can generate data according to a scale parameter given in terms of the number of products. The queries contain placeholders that are instantiated by actual values during the test phase.

### 5.2.1. *Experiment on Data Generation*

*Setup.* We used the two generators to create six data instances, denoted as BSBM-$s$-$g$, where $s \in \{1, 10, 100\}$ indicates the scale factor with respect to an initial data instance of 10000 products (produced by NTV), and $g \in \{\text{VIG}, \text{NTV}\}$ indicates the generator used to produce the instance.

*Results and Discussion.* Figure 4 shows the resources (time and memory) used by the two generators for creating the instances. For both generators the execution time grows approximately as the scale factor, which suggests that the generation of a single column value is in both cases independent from the size of the data instance to be generated. Observe that NTV is on average 5 times faster than VIG (in single-thread mode), but it also requires increasingly more memory as the amount of the data to generate increase, contrary to VIG that always requires the same amount of memory.

### 5.2.2. *Experiment on Query Evaluation*

We compare the execution times for the queries in the BSBM benchmark evaluated over the instances

[14]For data instances, generators, mappings, queries, etc., see `https://github.com/ontop/ontop-examples/tree/master/swj-2017-vig`

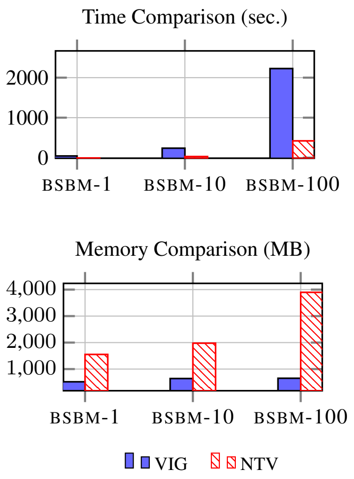


Fig. 4. Generation Time and Memory Comparison.

produced by VIG and NTV. Additionally, we here consider three additional data instances created with a random generator (RAND) that only considers database integrity constraints (primary and foreign keys) as similarity measures, ignoring the data statistics. This allows us to quantify the impact of the measures maintained by VIG on the task of approximating (w.r.t. the task of benchmarking) the data produced by NTV.

*Setup.* The experiment was run on a variation of the BSBM benchmark over the testing platform, the mappings, and the considered queries. We now briefly discuss and motivate the variations, before introducing the results.

The testing platform of the BSBM benchmark instantiates the queries with concrete values coming from binary configuration files produced by NTV. This does not allow a fair comparison between the three generators, because it is biased towards the specific values produced by NTV. Therefore, we reused the OBDA-Mixer of the NPD benchmark, which is independent from the specific generator used as it instantiates the queries only with values found in the provided database instance.

Another important difference regards the mapping component. The BSBM mapping contains some URI templates with two arguments where one of them is a unary primary key. This is commonly regarded as a bad practice in OBDA, as it is likely to introduce redundancies in terms of retrieved information. For instance, consider the template

```
bsbm-i:dataFSite/{publisher}/Reviewer{nr}
```

Table 7
Overview of the BSBM Experiment (D2RQ-MySQL).

| db | avg(ex_t) msec. | qmpH | avg(mix_t) msec. | [$\sigma$] |
|---|---|---|---|---|
| BSBM-1-NTV | 135446 | 2.95 | 1219012 | [229.436] |
| BSBM-1-VIG | 135444 | 2.95 | 1219000 | [43.8254] |
| BSBM-1-RAND | 135336 | 2.96 | 1218028 | [217.216] |
| BSBM-10-NTV | 140639 | 2.84 | 1265755 | [3344.42] |
| BSBM-10-VIG | 140821 | 2.84 | 1267391 | [557.378] |
| BSBM-10-RAND | 140855 | 2.84 | 1267694 | [2033.11] |
| BSBM-100-NTV | 424915 | 0.94 | 3824234 | [14976.1] |
| BSBM-100-VIG | 426015 | 0.94 | 3834135 | [2940.4] |
| BSBM-100-RAND | 411002 | 0.97 | 3699016 | [6463.91] |

Table 8
Overview of the BSBM Experiment (D2RQ-PostgreSQL).

| db | avg(ex_t) msec. | qmpH | avg(mix_t) msec. | [$\sigma$] |
|---|---|---|---|---|
| BSBM-1-NTV | 240185 | 1.67 | 2161668 | [17334.4] |
| BSBM-1-VIG | 246937 | 1.62 | 2222434 | [4435.64] |
| BSBM-1-RAND | 289572 | 1.38 | 2606150 | [12374.5] |
| BSBM-10-NTV | 431865 | 0.93 | 3886782 | [7287.9] |
| BSBM-10-VIG | 427559 | 0.94 | 3848029 | [2131.52] |
| BSBM-10-RAND | 426726 | 0.94 | 3840537 | [1704.36] |
| BSBM-100-NTV | 568777 | 0.70 | 5118991 | [69847.8] |
| BSBM-100-VIG | 543805 | 0.74 | 4894247 | [319.844] |
| BSBM-100-RAND | 540531 | 0.74 | 4864781 | [923.719] |

used to construct objects for the class `bsbm:Person`. The template has as arguments the primary key `nr` for the table `person`, plus an additional attribute `publisher`. Observe that, being `nr` a primary key, the information about the publisher does not contribute to the identification of specific persons. Additionally, the relation between persons and publishers is already realized in the mappings by a specific mapping assertion for the property `dc:publisher`. This mapping assertion poses a challenge to data generation, because query results are influenced by inclusion dependencies between binary tuples stored in different tables. VIG cannot correctly reproduce such inclusions, because it only supports inclusions (even not explicitly declared in the schema) between single columns. Observe that this problem would not be addressed even by supporting multi-attribute foreign keys, because such keys are not defined in the BSBM schema. For these reasons, we have changed the problematic URI template into a unary template by removing the redundant attribute `publisher`, so as to build individuals only out of primary keys. Observe that this change does not influence the semantics of the considered queries, nor their complexity.

We tested the 9 `SELECT` queries from the BSBM query set. We slightly modified two queries by relaxing an excessively restricting `FILTER` condition, so as to avoid empty results sets. We point out that this modification only slightly changes the size of the produced SQL translation, and that the modified queries are at least as hard as the original ones.

Table 9
Overview of the BSBM Experiment (Ontop-MySQL).

| db | avg(ex_t) msec. | qmpH | avg(mix_t) msec. | [$\sigma$] |
|---|---|---|---|---|
| BSBM-1-NTV | 2762 | 144.83 | 24856 | [25.1396] |
| BSBM-1-VIG | 2767 | 144.58 | 24899 | [224.129] |
| BSBM-1-RAND | 2677 | 149.40 | 24097 | [456.699] |
| BSBM-10-NTV | 4338 | 92.21 | 39041 | [373.56] |
| BSBM-10-VIG | 4159 | 96.17 | 37434 | [656.651] |
| BSBM-10-RAND | 3037 | 131.72 | 27331 | [235.793] |
| BSBM-100-NTV | 22471 | 17.80 | 202237 | [28799.8] |
| BSBM-100-VIG | 23328 | 17.15 | 209950 | [19442] |
| BSBM-100-RAND | 10020 | 39.92 | 90179 | [12347.7] |

Table 10
Overview of the BSBM Experiment (Ontop-PostgreSQL).

| db | avg(ex_t) msec. | qmpH | avg(mix_t) msec. | [$\sigma$] |
|---|---|---|---|---|
| BSBM-1-NTV | 2681 | 149.22 | 24125 | [143.818] |
| BSBM-1-VIG | 2649 | 151.02 | 23837 | [63.3219] |
| BSBM-1-RAND | 2654 | 150.69 | 23891 | [348.161] |
| BSBM-10-NTV | 3379 | 118.39 | 30409 | [827.423] |
| BSBM-10-VIG | 3234 | 123.68 | 29107 | [957.519] |
| BSBM-10-RAND | 3127 | 127.90 | 28147 | [654.745] |
| BSBM-100-NTV | 26222 | 15.25 | 236005 | [29750.7] |
| BSBM-100-VIG | 14128 | 28.31 | 127149 | [37339.8] |
| BSBM-100-RAND | 9924 | 40.31 | 89316 | [17959.8] |

*Results and Discussion.* Tables from 7 to 10 contain the results of the experiment in terms of various performance measures, namely the average execution time for the queries in mix (avg($ex\_t$)), the number of query mixes per hour ($qmpH$), the average mix time (avg($mix\_t$)), and the standard deviation calculated over the mix times ([$\sigma$]). We observe that the measured performance for queries executed over the instances produced by VIG is very close to the measured performance for queries executed over the instances (of comparable size) produced by NTV. This confirms the hypothesis that VIG can produce data that are of acceptable quality for benchmarking in the BSBM setting. Moreover, for the tests with Ontop (Tables 9 and 10), we observe that VIG performs substantially better than RAND, as the tests over the RAND instances generally run twice faster than the tests over the instances generated by VIG or NTV. However, the tests over D2RQ (Tables 7 and 8) display no substantial difference between the measured performance for queries executed over the instances produced by RAND and the other instances, despite the extremely naive generation strategy. We realized that the reason for this odd behavior is the fact that D2RQ is substantially (two orders of magnitude) slower than Ontop when tested against the BSBM benchmark: in D2RQ many queries time out, flattening the overall mix time. The reason of the performance discrepancy between the two OBDA systems is due to the ability of Ontop to optimize the produced `SQL` translations by exploiting database dependencies such as primary and foreign keys [5]. These optimizations, not performed by D2RQ, are apparently very beneficial in the BSBM setting.

#### 5.2.3. Experiment on Predicate Growth

*Setup.* In this experiment we evaluate how scaling with VIG affects the growth of classes and properties in the BSBM ontology. In particular, we expect this growth to be consistent with the growth observed on the data instances produced by NTV.

To perform this experiment, we have created a query for each class $C$ and property $P$ in the ontology retrieving all the individuals or tuples of individuals belonging to $C$ or $P$. We evaluate each such query on the data instances generated by VIG and those generated by NTV, expecting that the number of obtained results is similar for both generators. In total, we have checked the growth of 8 classes, 10 object properties, and 30 data properties.

*Results and Discussion.* Table 11 shows the deviation, in terms of number of elements for each predicate (class, object property, or data property) in the ontology, between the instances generated by VIG and those generated by NTV. The column avg(dev) reports the average percent deviation. The last two columns report respectively the absolute number and relative percentage of predicates for which the deviation was greater than 5%. We observe that the deviation for predicates growth is inferior to 5% for the majority of classes and properties in the ontology. The few outliers are due to some predicates that are built from tables that NTV, contrary to VIG, does not scale according to the scale factor. These predicates are the two classes `ProductFeature` and `ProductType`, and the 4 properties `productFeature`, `comment`, `label` and `subClassOf`[15].

Table 11
Predicates Growth Comparison.

| type-db-scale | avg(dev) | dev > 5% (#) | dev > 5% (%) |
|---|---|---|---|
| CLASS-BSBM-1 | 0% | 0 | 0% |
| CLASS-BSBM-10 | 23.72% | 2 | 25% |
| CLASS-BSBM-100 | 250.74% | 2 | 25% |
| OBJ-BSBM-1 | 0% | 0 | 0% |
| OBJ-BSBM-10 | 7.46% | 2 | 20% |
| OBJ-BSBM-100 | 82.35% | 2 | 20% |
| DATA-BSBM-1 | < 0.01% | 0 | 0% |
| DATA-BSBM-10 | 2.84% | 2 | 6.67% |
| DATA-BSBM-100 | 5.74% | 2 | 6.67% |

### 5.3. DBLP Experiment

The DBLP computer science bibliography[16] is an on-line reference for bibliographic information on major computer science publications. The data is released as open data under the ODC-BY 1.0 license. The DBLP++ data set is an enhancement of DBLP with additional keywords and abstracts. The DBLP++ data is stored in a MySQL database and can be accessed through a SPARQL endpoint powered by D2RQ. The database dump, the mapping and configuration of D2RQ are published by the DBLP team[17]. The MySQL database dump is provided as a gzipped file of 564MB, containing 1.9M authors, 3.6M publications, and 12.3M author-publication relations. We have also converted the dump into PostgreSQL.

#### 5.3.1. Experiment on Query Evaluation

*Setup.* We used VIG to scale the original DBLP data instance (DBLP), and produced three instances of scaling factors 1, 3, and 5 (named DBLP-$s$-VIG, $s \in \{1,3,5\}$, respectively). We also used RAND to scale the original DBLP data instance of a scaling factor of 1 (DBLP-1-RAND).

[15] An object property defined under the BSBM namespace.
[16] `http://dblp.uni-trier.de/`
[17] `http://dblp.l3s.de/dblp++.php`

Table 12
Overview of the DBLP Experiment (D2RQ-MySQL).

| db | avg(ex_t) msec. | qmpH | avg(mix_t) msec. | [$\sigma$] |
|---|---|---|---|---|
| DBLP | 180265 | 1.25 | 2884246 | [589596] |
| DBLP-1-VIG | 154780 | 1.45 | 2476480 | [10695.5] |
| DBLP-1-RAND | 2209 | 101.86 | 35341 | [218.634] |
| DBLP-3-VIG | 157348 | 1.43 | 2517575 | [37772.8] |
| DBLP-5-VIG | 167884 | 1.34 | 2686144 | [61017.3] |

Table 13
Overview of the DBLP Experiment (D2RQ-PostgreSQL).

| db | avg(ex_t) msec. | qmpH | avg(mix_t) msec. | [$\sigma$] |
|---|---|---|---|---|
| DBLP | 320145 | 0.70 | 5122325 | [90711.1] |
| DBLP-1-VIG | 155757 | 1.44 | 2492108 | [589903] |
| DBLP-1-RAND | 6702 | 33.57 | 107231 | [1501.64] |
| DBLP-3-VIG | 294028 | 0.77 | 4704454 | [666089] |
| DBLP-5-VIG | 409211 | 0.55 | 6547440 | [172190] |

Table 14
Overview of the DBLP Experiment (Ontop-MySQL).

| db | avg(ex_t) msec. | qmpH | avg(mix_t) msec. | [$\sigma$] |
|---|---|---|---|---|
| DBLP | 173978 | 1.29 | 2783649 | [5028.72] |
| DBLP-1-VIG | 178983 | 1.26 | 2863728 | [8409.02] |
| DBLP-1-RAND | 48544 | 4.63 | 776712 | [14737] |
| DBLP-3-VIG | 259885 | 0.87 | 4158162 | [19578.3] |
| DBLP-5-VIG | 283950 | 0.79 | 4543207 | [72154] |

Table 15
Overview of the DBLP Experiment (Ontop-PostgreSQL).

| db | avg(ex_t) msec. | qmpH | avg(mix_t) msec. | [$\sigma$] |
|---|---|---|---|---|
| DBLP | 200159 | 1.12 | 3202546 | [980.228] |
| DBLP-1-VIG | 213306 | 1.05 | 3412889 | [391.852] |
| DBLP-1-RAND | 120924 | 1.86 | 1934778 | [6979.83] |
| DBLP-3-VIG | 317992 | 0.71 | 5087885 | [9867.96] |
| DBLP-5-VIG | 344435 | 0.65 | 5510988 | [23256.4] |

We manually crafted a set of 16 queries resembling some real-world information needs from DBLP (e.g., "list all the authors of a journal", "list all coauthors of an author", "list all publications directly referencing a publication from one author", or "list publications indirectly referencing[18] a publication from one author"), and evaluated them against the instances produced by VIG and RAND, and against the original DBLP instance. For the experiments we used the mapping file provided by the DBLP team, written in the D2RQ mappings syntax, and an equivalent version written in the Ontop syntax, obtained with the help of a converter[19].

*Results and Discussion.* Tables from 12 to 15 contain the results of the experiments. We observe that the measured performances for query evaluation over the instances DBLP and DBLP-1-VIG are relatively close. This confirms that VIG is able to generate data of acceptable quality for benchmarking in the DBLP setting. We point out that, contrary to synthetically generated instances from BSBM, the source data instance in the DBLP experiment is a real-world instance. We also observe that queries evaluated over the instance DBLP-1-RAND, obtained with the random generator, have substantially different execution times from the ones evaluated in either DBLP or DBLP-1-VIG (up to 2 orders of magnitude faster for D2RQ/MySQL, in Table 12). Interestingly, the difference is not so extreme for the tests with Ontop. In fact, Tables 14 and 15 show that Ontop performs significantly slower than D2RQ on the instance DBLP-1-RAND. By manually examining the generated SQL queries and their results, we discovered that only 2 queries out of 16 return a non-empty result when evaluated over the random instance DBLP-1-RAND by both OBDA systems. Hence 14 queries are executed extremely fast. The remaining two queries run slow in Ontop, whereas they are executed efficiently by D2RQ. These two queries have a similar shape, and they use a combination of the `DISTINCT` and `LIMIT` modifiers. We found out that Ontop handles such queries in a different and less efficient way than D2RQ, which explains the difference in the observed performance.

Since already at the scale factor 1, RAND behaves significantly different from the original data and VIG, for larger scale factors 3 and 5, we only evaluated the queries over data generated by VIG. Looking at three instances DBLP-$s$-VIG, $s \in \{1, 3, 5\}$, we observe that the average execution time and the mix time grow roughly proportionally to the scale factor.

[18]Under a bound on the length of the chain of indirect references.
[19]`https://github.com/RMLio/D2RQ_to_R2RML`

### 5.3.2. Experiment on Predicate Growth

*Setup.* In this experiment we evaluate how scaling with VIG affects the growth of classes and properties in the DBLP ontology. In particular, we compare the original DBLP dataset and DBLP-1-VIG. The test is set up analogously to the BSBM case. In total, we have checked the growth of 55 classes, 44 object properties, and 30 data properties.

*Results and Discussion.* Table 16 shows the deviation, in terms of number of elements for each predicate (class, object property, or data property) in the ontology, between DBLP and DBLP-1-VIG. Similarly to what we observed for BSBM, the average deviation for classes is strongly influenced by a few outliers. We observe that this deviation is inferior to $5\%$ for the majority of classes and properties in the ontology. The reason of the deviation in DBLP is due to the fact that VIG is assuming a uniform distribution of the data whereas real-world instances do not always follow this assumption. For instance, some productive authors have hundreds of publications but the majority of the authors only have a few publications.

## 5.4. NPD Experiment

The NPD Benchmark [17] is a benchmark for OBDA systems based on the the Norwegian Petroleum Directorate (NPD) FactPages[20]. The benchmark comes with an ontology, a set of mappings containing thousands of mapping assertions, and 31 SPARQL queries that resemble information needs from the users of the NPD Factpages. Contrary to the previous experiments, this experiment allows for testing the impact of the mappings analysis on the quality of the scaled data, since NPD is the only setting (among the considered ones) in which we have a structured ontology and complex mappings. For this experiment we have used only Ontop, as we did not manage to load the NPD mappings into D2RQ. Moreover, we do not include a comparison on NPD against the random data generator, as this aspect has already been discussed in [17].

[20]http://factpages.npd.no/factpages.

Table 16
Predicates Growth Comparison.

| type-db-scale | avg(dev) | dev > 5% (#) | dev > 5% (%) |
|---|---|---|---|
| CLASS-DBLP-1-VIG | 142.2% | 8 | 14.54% |
| OBJ-DBLP-1-VIG | 0.48% | 1 | 2.32% |
| DATA-DBLP-1-VIG | 0% | 0 | 0% |

Table 17
Overview of the NPD Experiment (NPD-MySQL).

| db | avg(ex_time) msec. | qmpH | avg(mix_time) [$\sigma$] msec. |
|---|---|---|---|
| NPD | 50263 | 2.98 | 1206560 [686.582] |
| NPD-1-VIG | 50235 | 2.99 | 1205944 [491.685] |
| NPD-5-VIG | 51052 | 2.94 | 1226204 [1158.09] |
| NPD-10-VIG | 52204 | 2.87 | 1254703 [5595.96] |
| NPD-50-VIG | 64467 | 2.31 | 1559613 [49349.4] |
| NPD-100-VIG | 86769 | 1.71 | 2108334 [92309.3] |
| NPD-500-VIG | 229213 | 0.64 | 5663050 [61835.2] |

### 5.4.1. Experiment on Query Evaluation

*Setup.* We used VIG to scale the original NPD data instance (NPD), and produced six instances of scaling factors 1, 5, 10, 50, 100, and 500 (named NPD-$s$-VIG, $s \in \{1, 5, 10, 50, 100, 500\}$, respectively).

For the test, we used the $24$ queries from the NPD benchmark that are supported by Ontop (i.e., all queries without aggregation operators), and evaluated them against the instances produced by VIG and against the original NPD instance. For the experiments we used the mapping file included with the NPD benchmark.

*Results and Discussion.* Tables 17 and 18 contain the results of the experiments. We observe that the measured performances for query evaluation over the instances NPD and NPD-1-VIG almost coincide, in both MySQL and PostgreSQL. This confirms that VIG is able to generate data of acceptable quality for benchmarking in the NPD setting. As already done with DBLP, we point out also here that, contrary to synthetically generated instances from BSBM, the source data instance in the NPD experiment is a real-world instance.

Looking at the other scaled instances, we observe that the average execution time and the mix time over the evaluations for PostgreSQL grow roughly proportionally to the scale factor. This is not the case for the MySQL evaluations, due to the presence of a few queries which timeout and flatten the overall mix times.

### 5.4.2. Experiment on Predicate Growth

*Setup.* In this experiment we evaluate how scaling with VIG affects the growth of classes and properties in the NPD ontology. In particular, we compare the original NPD dataset and NPD-1-VIG. The test is set up

Table 18
Overview of the NPD Experiment (NPD-PostgreSQL).

| db | avg(ex_time) msec. | qmpH | avg(mix_time) [$\sigma$] msec. |
|---|---|---|---|
| NPD | 2624 | 57.00 | 63158 [873.707] |
| NPD-1-VIG | 2842 | 52.58 | 68467 [1559.15] |
| NPD-5-VIG | 3694 | 40.30 | 89321 [2258.51] |
| NPD-10-VIG | 7281 | 20.43 | 176197 [4413.34] |
| NPD-50-VIG | 29790 | 4.97 | 724506 [32490.3] |
| NPD-100-VIG | 53344 | 2.78 | 1297169 [41639.2] |
| NPD-500-VIG | 169653 | 0.64 | 5587795 [2.00471e+06] |

Table 19
Predicates Growth Comparison.

| type-db-scale | avg(dev) | dev > 5% (#) | dev > 5% (%) |
|---|---|---|---|
| CLASS-NPD-1-VIG | 236.43% | 70 | 20.42% |
| OBJ-NPD-1-VIG | 63.68% | 20 | 14.0% |
| DATA-NPD-1-VIG | 26.93% | 44 | 18.56% |

analogously to the BSBM and DBLP cases. In total, we have checked the growth of 343 classes, 142 object properties, and 238 data properties.

*Results and Discussion.* Table 19 shows the deviation, in terms of number of elements for each predicate (class, object property, or data property) in the ontology, between NPD and NPD-1-VIG. Similarly to what we observed for BSBM and DBLP, the average deviation for classes is strongly influenced by a few outliers. We observe that this deviation is inferior to 5% for the majority of classes and properties in the ontology. The reason for the deviation in NPD is similar to what we have already encountered in DBLP, and it is due to the complex data distributions in real world instances, which are not captured by VIG.

*5.4.3. Experiment on The Use of Domain Information*
*Setup.* The query discussed in our running example is at the basis of the three hardest real-world queries in the NPD Benchmark, namely queries 6, 11, and 12. In this section we compare these three queries on two modalities of VIG: one in which only the input database is taken as input (DB mode), and for which the columns cluster analysis is not performed, and the one (OBDA mode) discussed in this paper where the mapping is also taken into account.

*Results and Discussion.* Table 20 contains the selectivities (i.e., number of results) of all four possible

Table 20
Selectivity Analysis.

| joins | NPD | NPD-1 | | NPD-5 | | NPD-50 | |
|---|---|---|---|---|---|---|---|
| | | DB | OBDA | DB | OBDA | DB | OBDA |
| $\|sw \bowtie ew\|$ | 0 | 841 | 0 | 5046 | 0 | 42891 | 0 |
| $\|sw \bowtie dw\|$ | 0 | 841 | 0 | 5046 | 0 | 42891 | 0 |
| $\|ew \bowtie dw\|$ | 0 | 1560 | 0 | 9344 | 0 | 79814 | 0 |
| $\|sw \bowtie ew \bowtie dw\|$ | 0 | 841 | 0 | 5046 | 0 | 42891 | 0 |

Table 21
Evaluations for queries 6, 11, and 12.

| query | NPD | NPD-1 | | NPD-5 | | NPD-50 | |
|---|---|---|---|---|---|---|---|
| | | DB | OBDA | DB | OBDA | DB | OBDA |
| q6 | 787 | 597 | 456 | 10689 | 1494 | 17009 | 6961 |
| q11 | 661 | 1020 | 364 | 2647 | 1487 | 37229 | 15807 |
| q12 | 1190 | 2926 | 714 | 8059 | 3363 | 38726 | 17830 |

joins between the three tables `shallow_wellbore` (abbreviated sw), `exploration_wellbore` (abbreviated ew), and `development_wellbore` (abbreviated dw), over the original NPD dataset as well as its scaled versions of factors 1, 5, and 50. Observe that the instances created through the OBDA mode correctly produces zero selectivities by analyzing the mappings as described in Section 4.1. On the contrary, the instances created through the DB mode produce joins of non-zero selectivities. This fact, together with the mapping definitions of the NPD benchmark (in Table 3 we show the portion for the classes ExpWellbore and ShWellbore; the class DevWellbore is mapped in a similar way) produce a violation of the disjointness constraints between these classes in the NPD ontology.

Table 21 shows the impact of the wrong selectivities on the performance (response time in milliseconds) of evaluation for the queries under consideration. Observe that the performance measured over the DB instances differ sensibly from the one measured over OBDA instances, or over the original NPD instance. This is due to the higher costs for the join operations in DB instances, which in turn derive from the wrong selectivities discussed in the previous paragraph.

## 6. Discussions and Limitations

In our experiments, we observed that the performance for query answering over the instances generated by VIG is comparable to the performance for query answering over the original data instances. This observation suggests that the data generated by VIG is suitable for benchmarking OBDA systems in the considered settings.

However, the encouraging conclusion will not apply to every setting. In fact, observe that VIG considers only a limited set of similarity measures, and that the produced instances that are similar in terms of these measures might be not enough to guarantee reliable performance analyses. For instance, we have already discussed how VIG is not able to reproduce constraints such as multi-attribute foreign keys or non-uniform data distributions, or how implicit inclusion dependencies between tuples are not being considered (c.f. Section 5.2.2). Thus, although we show here how VIG seems to suffice for BSBM and DBLP (under our assumptions for queries, mappings, and testing platform), we expect it not to perform as good in more complex scenarios, where the non-supported measures become significant. Indeed, we already observed in Table 16 of the DBLP experiment that the data distribution is playing an important role for generating instances of some classes in the ontology.

Moreover, an intrinsic weakness of VIG, and of the scaling approach in general, is that it only considers a *single* source data instance: in case certain measures depend on the size of the instance, as it seems to be the case for two classes and properties in Table 11, then the scaled instances might significantly diverge from the real ones.

## 6.1. Discussion on Multi-attribute Foreign Keys

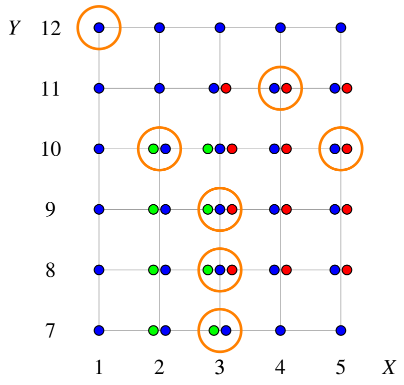
We have discussed how VIG can produce data that retain certain statistics observed on the initial seed of data, while complying with schema constraints such as primary keys and single-attribute foreign keys. In this section we discuss the impact of multi-attribute foreign keys, and why it is non-trivial to satisfy this kind of constraint while retaining the discussed statistics. We carry out our discussion by means of an example.

**Example 6.1.** Consider a database schema $\Sigma$ with three tables $T_1$, $T_2$, and $T_3$, where each table $T_i$ contains columns $X_i$ and $Y_i$, for $i \in \{1,2,3\}$. Suppose that $\Sigma$ defines a primary-key constraint $K_i := (X_i, Y_i)$ on each table, and the following multi-attribute foreign-key dependencies:

$$fk_1 := K_1 \subseteq K_3, \qquad fk_2 := K_2 \subseteq K_3.$$

For each $K_i$, with $i \in \{1,2,3\}$, we denote by $\mathrm{ints}(K_i)$ the pair $(\mathrm{ints}(X_i), \mathrm{ints}(Y_i))$, indicating that the first element of a tuple in $K_i$ should belong to the intervals $\mathrm{ints}(X_i)$, and that the second element should belong to the intervals $\mathrm{ints}(Y_i)$.

Fig. 5. Multi-attribute Foreign Key Satisfaction. The $X$ axis lists all allowed values for the attributes $X_i$, i.e., all values in $\bigcup_i \mathrm{ints}(X_i)$. Similarly, the $Y$ axis lists all allowed values for the attributes $Y_i$. Each position $(x, y)$ in the grid is marked with the colors of the keys in which the tuple $(x, y)$ is allowed. For instance, position $(1, 7)$ is a valid tuple for $K_3$, but not for $K_1$ or for $K_2$. Orange circles represent a possible assignment of positions satisfying all the constraints specified in Example 6.1.



Suppose that, after an analysis phase over a database instance $\mathcal{D}$, VIG has produced the following intervals:

$$\begin{aligned}\mathrm{ints}(K_1) &= (\{[3,5]\}, \{[8,11]\}),\\ \mathrm{ints}(K_2) &= (\{[2,3]\}, \{[7,10]\}),\\ \mathrm{ints}(K_3) &= (\{[1,5]\}, \{[7,12]\}).\end{aligned}$$

Let us call $\mathcal{D}'$ the database instance to be produced by VIG, and suppose that, after the analysis phase, VIG has established the following statistics:

$$\begin{aligned}\mathrm{size}(T_1, \mathcal{D}') &= 4,\\ \mathrm{size}(T_2, \mathcal{D}') &= 4,\\ \mathrm{size}(T_3, \mathcal{D}') &= 7.\end{aligned}$$

Then, the problem of satisfying the dependencies $fk_1$ and $fk_2$ in the generated instance $\mathcal{D}'$, under these assumptions, reduces to the problem of finding 7 positions in the bi-dimensional space from Figure 5, where each of these positions represents a tuple for $K_3$. Observe that positions have to be picked so that 4 of them represent tuples for $K_2$, 4 of them represent tuples for $K_1$, and exactly one position represents a tuple that is shared between $K_1$, $K_2$, and $K_3$. ■

In general, if there are $k$ attributes used for foreign key constraints, then this is a search problem in a $k$-dimensional space. Observe that, even if this search could be done in polynomial time, it would still be dramatically slower than the current implementation of VIG, which only requires constant-time and no memory accesses to generate a tuple.

## 7. Related Work

In this section we discuss the relation between VIG and other data scalers, as it makes little sense to compare it to classic data generators used in OBDA benchmarks as, for instance, the one found in the *Texas Benchmark* [21].

UpSizeR [26] replicates two kinds of distributions observed on the values for the key columns, called *joint degree distribution* and *joint distribution over co-clusters*[22]. However, this requires several assumptions to be made on the $\Sigma$, for instance tables can have at most two foreign keys, primary keys cannot be multi-attribute, etc. Moreover, generating values for the foreign keys require reading of previously generated values, which is not required in VIG. *Rex* [3] is an approach closely related to UpSizeR, but that it is easier to configure than UpSizeR. This approach provides, through the use of dictionaries, a better handling of the content for non-key columns than UpSizeR or VIG. The limitations mentioned for UpSizeR apply for this system as well.

In terms of similarity measures, the approach closest to VIG is *RSGen* [25], that also considers measures like NULL ratios or number of distinct values. Moreover, values are generated according to a uniform distribution, as in VIG. However, the approach only works on numerical data types, and it seems not to support multi-attribute primary keys. A related approach, but with the ability of generating data for non-numerical fields, has been proposed in [23]. Notably, this approach is able to produce *realistic* text fields by relying on machine-learning techniques based on Markov chains.

In *RDF graph scaling* [22], an additional parameter, called *node degree scaling factor*, is provided as input to the scaler. The approach is able to replicate the phenomena of *densification* that have been observed for certain types of networks. However, the quality of the data generated by an RDF scaler and by VIG is not directly comparable: RDF scaling is able to exploit more graph-specific statistics because it is not constrained by any schema and it directly generates an RDF graph; instead, VIG has to be compliant with the schema constraints of the underlying database and the generation of the RDF graph depends on the mappings. On the positive side, these OBDA specific inputs (i.e., database constraints and mappings) improve the quality of the generated data; on the negative side, it is unclear how to incorporate graph-specific statistics in this setting. In addition, we observe that the general RDF graph scaling approach is not suitable for benchmarking OBDA systems, where the goal of data scaling is to obtain a scaled-up database instance. In fact, the problem of obtaining a database instance, given an RDF graph (produced by an RDF graph scaler) and a set of mappings, corresponds to that of generating a database instance given the extensions of database views, where the queries in the source part of the mappings act as view definitions. This problem is in turn equivalent to the view update problem [9], which is known to be challenging and actually solvable only for a very restricted class of queries used in the mappings.

Observe that all the approaches above do not consider ontologies or mappings. Therefore, many measures important in a context with mappings and ontologies and discussed here, like selectivities for joins in a co-cluster, class disjointness, or reuse of values for fixed-domain columns, are not taken into consideration in such approaches. This leads to problems like the one we discussed through our running example, and for which we showed in Section 5 how it affects the benchmarking analysis.

## 8. Conclusion and Development Plan

In this work we presented VIG, a data-scaler for OBDA benchmarks. VIG integrates some of the measures used by database query optimizers and existing data scalers with OBDA-specific measures, in order to deliver a better data generation in the context of OBDA benchmarks.

We have evaluated VIG in the task of generating data for the BSBM, DBLP, and NPD benchmarks. In BSBM and DBLP, we measured how *similar* the data produced by VIG is to the one produced by the native BSBM generator and the original DBLP data instance, obtaining encouraging results. In the NPD benchmark, we provided an empirical evaluation of the impact that the most distinguished feature of VIG, namely the mappings analysis, has on the shape of the produced in-

[21] `http://obda-benchmark.org/`

[22] The notion of co-cluster has nothing to do with the notion of columns-cluster introduced here.

stances, and how it affects the measured performance of benchmark queries.

The current work plan is to enrich the quality of the generated data by adding support for multi-attribute foreign keys, joint-degree and value distributions, and intra-row correlations (e.g., objects from "Suspended Wellbore" might not have a "Completion Year"). Unfortunately, we expect that some of these extensions will conflict with the current feature of constant time for the generation of tuples. In fact, they might require access to previously generated tuples in order to correctly compute new tuples (e.g., to take into account joint-degree distribution [26]).

A related problem is how to extend the notion of "scaling" to the other components forming an input for the OBDA system, like the mappings, the ontology, or the queries. We see this as an interesting research problem to be addressed in the future.

*Acknowledgments.* This research has been partially supported by the projects "Ontology-based analysis of temporal and streaming data" (OBATS) and "High Quality Data Integration with Ontologies" (QUADRO) funded through the 2017 and 2018 research budgets of the Free University of Bozen-Bolzano, and by the project "South Tyrol Longitudinal study on Longevity and Ageing" (STyLoLa), funded through the 2017 Interdisciplinary Call issued by the Research Committee of the Free University of Bozen-Bolzano.

## References

[1] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA, 2003.

[2] Christian Bizer and Andreas Schultz. The Berlin SPARQL benchmark. *Int. J. on Semantic Web and Information Systems*, 5(2):1–24, 2009. doi:10.4018/jswis.2009040101.

[3] Teodora Sandra Buda, Thomas Cerqueus, Cristian Grava, and John Murphy. Rex: Representative extrapolating relational databases. *Information Systems*, 67:83–99, 2017. doi:10.1016/j.is.2017.03.001.

[4] Damian Bursztyn, François Goasdoué, and Ioana Manolescu. Teaching an RDBMS about ontological constraints. *Proc. of the VLDB Endowment*, 9(12):1161–1172, 2016. doi:10.14778/2994509.2994532.

[5] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering SPARQL queries over relational databases. *Semantic Web J.*, 8(3):471–487, 2017. doi:10.3233/SW-160217.

[6] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, and Riccardo Rosati. Ontologies and databases: The *DL-Lite* approach. In Sergio Tessaris and Enrico Franconi, editors, *Reasoning Web: Semantic Technologies for Informations Systems – 5th Int. Summer School Tutorial Lectures (RW), Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*, volume 5689 of *Lecture Notes in Computer Science*, pages 255–356. Springer, New York, NY, USA, 2009. doi:10.1007/978-3-642-03754-2_7.

[7] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. Linking data to ontologies: The description logic *DL-Lite*$_{\mathcal{A}}$. In *Proc. of the 2nd Int. Workshop on OWL: Experiences and Directions (OWLED), Athens, Georgia, USA, November 10-11, 2006*, volume 216 of *CEUR Workshop Proceedings*, `http://ceur-ws.org/`, 2006.

[8] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. of Automated Reasoning*, 39(3):385–429, 2007. doi:10.1007/s10817-007-9078-x.

[9] Stavros S. Cosmadakis and Christos H. Papadimitriou. Updates of relational views. *J. of the ACM*, 31(4):742–760, 1984. doi:10.1145/1634.1887.

[10] Souripriya Das, Seema Sundara, and Richard Cyganiak. R2RML: RDB to RDF mapping language. W3C Recommendation, World Wide Web Consortium, September 2012. Available at `http://www.w3.org/TR/r2rml/`.

[11] Floriana Di Pinto, Domenico Lembo, Maurizio Lenzerini, Riccardo Mancini, Antonella Poggi, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. Optimizing query rewriting in ontology-based data access. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 561–572, New York, NY, USA, 2013. ACM Press. doi:10.1145/2452376.2452441.

[12] Jörg Diederich, Wolf-Tilo Balke, and Uwe Thaden. Demonstrating the semantic GrowBag: Automatically creating topic facets for FacetedDBLP. In *Proc. of the 7th ACM/IEEE Joint Conf. on Digital Libraries (JCDL)*, page 505, New York, NY, USA, 2007. ACM Press. doi:10.1145/1255175.1255305.

[13] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *Proc. of the 15th ACM Int. Conf. on Management of Data (SIGMOD), Minneapolis, Minnesota, USA, May 24-27, 1994*, pages 243–252, New York, NY, USA, 1994. ACM Press. doi:10.1145/191839.191886.

[14] Steve Harris and Andy Seaborne. SPARQL 1.1 query language. W3C Recommendation, World Wide Web Consortium, March 2013. Available at `http://www.w3.org/TR/sparql11-query`.

[15] Dag Hovland, Davide Lanti, Martin Rezk, and Guohui Xiao. OBDA constraints for effective query answering. In *Proc. of the 10th Int. Symp. on Rule Technologies: Research, Tools, and Applications (RuleML), Stony Brook, NY, USA, July 6-9, 2016*, volume 9718 of *Lecture Notes in Computer Science*, pages 269–286, New York, NY, USA, 2016. Springer. doi:10.1007/978-3-319-42019-6_18.

[16] Roman Kontchakov, Martin Rezk, Mariano Rodriguez-Muro, Guohui Xiao, and Michael Zakharyaschev. Answering SPARQL queries over databases under OWL 2 QL entailment regime. In *Proc. of the 13th Int. Semantic Web Conf. (ISWC), Riva del Garda, Italy, October 19-23, 2014*, volume 8796 of *Lecture Notes in Computer Science*, pages 552–567, New York, NY, USA, 2014. Springer. doi:10.1007/978-3-319-

11964-9_35.

[17] Davide Lanti, Martin Rezk, Guohui Xiao, and Diego Calvanese. The NPD benchmark: Reality check for OBDA systems. In *Proc. of the 18th Int. Conf. on Extending Database Technology (EDBT), Brussels, Belgium, March 23-27, 2015*, pages 617–628. OpenProceedings.org, 2015. doi:10.5441/002/edbt.2015.62.

[18] Davide Lanti, Guohui Xiao, and Diego Calvanese. Fast and simple data scaling for OBDA benchmarks. In *Proc. of the Workshop on Benchmarking Linked Data (BLINK)*, volume 1700 of *CEUR Workshop Proceedings,* `http://ceur-ws.org/`, 2016.

[19] Davide Lanti, Guohui Xiao, and Diego Calvanese. Cost-driven ontology-based data access. In *Proc. of the 16th Int. Semantic Web Conf. (ISWC), Vienna, Austria, October 21-25, 2017*, volume 10587 of *Lecture Notes in Computer Science*, pages 452–470, New York, NY, USA, 2017. Springer. doi:10.1007/978-3-319-68288-4_27.

[20] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. OWL 2 Web Ontology Language profiles (second edition). W3C Recommendation, World Wide Web Consortium, December 2012. Available at `http://www.w3.org/TR/owl2-profiles/`.

[21] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2015. Available at `http://www.choco-solver.org/`.

[22] Shi Qiao and Z. Meral Özsoyoğlu. RBench: Application-specific RDF benchmarking. In *Proc. of the 36th ACM Int. Conf. on Management of Data (SIGMOD), Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1825–1838, New York, NY, USA, 2015. ACM Press. doi:10.1145/2723372.2746479.

[23] Tilmann Rabl, Manuel Danisch, Michael Frank, Sebastian Schindler, and Hans-Arno Jacobsen. Just can't get enough: Synthesizing big data. In *Proc. of the 36th ACM Int. Conf. on Management of Data (SIGMOD), Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1457–1462, New York, NY, USA, 2015. ACM Press. doi:10.1145/2723372.2735378.

[24] Juan F. Sequeda, Marcelo Arenas, and Daniel P. Miranker. OBDA: Query rewriting or materialization? In practice, both! In *Proc. of the 13th Int. Semantic Web Conf. (ISWC), Riva del Garda, Italy, October 19-23, 2014*, volume 8796 of *Lecture Notes in Computer Science*, pages 535–551, New York, NY, USA, 2014. Springer. doi:10.1007/978-3-319-11964-9_34.

[25] Entong Shen and Lyublena Antova. Reversing statistics for scalable test databases generation. In *Proc. of the 6th Int. Workshop on Testing Database Systems (DBTest 2013)*, pages 7:1–7:6, New York, NY, USA, 2013. ACM Press. doi:10.1145/2479440.2479445.

[26] Y.C. Tay, Bing Tian Dai, Daniel T. Wang, Eldora Y. Sun, Yong Lin, and Yuting Lin. UpSizeR: Synthetically scaling an empirical relational database. *Information Systems*, 38(8):1168–1183, 2013. doi:10.1016/j.is.2013.07.004.

[27] Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, and Michael Zakharyaschev. Ontology-based data access: A survey. In *Proc. of the 27th Int. Joint Conf. on Artificial Intelligence (IJCAI), July 13-19, 2018, Stockholm, Sweden*, pages 5511–5519, 2018. doi:10.24963/ijcai.2018/777.

[28] Guohui Xiao, Roman Kontchakov, Benjamin Cogrel, Diego Calvanese, and Elena Botoeva. Efficient handling of SPARQL optional for OBDA. In *Proc. of the 17th Int. Semantic Web Conf. (ISWC), Monterey, US, Octotober 8-12, 2018*, New York, NY, USA, 2018. doi:10.1007/978-3-030-00671-6_20.