Semantic Web 0 (0) 1
IOS Press

VIG: Data Scaling for OBDA Benchmarks

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

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 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 read-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] 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 over which SPARQL queries can be posed. This solution is realized through *mappings* that link classes and properties in the ontology to queries over the database.

Proper benchmarking of query answering systems, such as OBDA systems, requires scalability analyses taking into account for 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 of 80 columns in [14]). 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 totally different and present different challenges [14].

Data scaling [22] 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, that should be *similar* to the one 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, that 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 easily imported by any relational database system.

VIG is a Java implementation licensed under Apache 2.0, and its source code is available on GitHub in the form of a Maven project [16]. The code is maintained by the Ontop team at the Free University of Bozen-Bolzano, and it comes with documentation in the form of Wiki pages.

To evaluate VIG, we have carried 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 read-world data, ad-hoc data generators and random data generators. In numbers, we ran in total 6698 queries over 35 database instances, among which 2 (DBLP and NPD) are real-world data, 3 are original synthetic data from BSBM, and the rest are generated by VIG in different modes (random, DB only, and full OBDA). The encouraging results over BSBM and DBLP 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 NPD dataset demonstrate the benefits of the mappings analysis on the quality of the scaled 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. Sections 6 and 7 contain related work and conclusions, respectively.

This paper is a significant extension of a preliminary version of this work presented in the Workshop on Benchmarking Linked Data Workshop (BLINK) 2016 [15], that only evaluated VIG over the two datasets NPD and BSBM with Ontop and MySQL.

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 in OBDA is OWL 2 QL [17]. For the sake of conciseness, we consider here its mathematical underpinning *DL-Lite*$_{\mathcal{R}}$ [8]. Table 1 shows a portion of the ontology from the NPD benchmark, which is the foundation block of our running example.

The W3C standard query language in OBDA is SPARQL [12], with queries evaluated under the OWL 2 QL entailment regime [13]. Intuitively, under these semantics each basic graph pattern (BGP) can be seen as a single conjunctive query (CQ) without existentially quantified variables. As in our examples we will only refer to SPARQL queries containing exactly one BGP, we will 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 [9], 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 $X(\vec{f}, \vec{x}) \leftarrow \mathrm{conj}(\vec{y})$, consisting of a *target* part $X(\vec{f}, \vec{x})$, which is an atom over function symbols $\vec{f}$ (also called *templates*) and variables $\vec{x} \subseteq \vec{y}$, and a *source* part $\mathrm{conj}(\vec{y})$, which is a CQ whose output variables are $\vec{y}$. We say that $\mathfrak{m}$ *defines the predicate X* if *X* is in the target of $\mathfrak{m}$. A *basic mapping* is a mapping whose source part contains exactly one atom. Table 3 contains the mappings for our running example, as well as a short description of how these mappings are used in order to create a (virtual) set of *assertions*.

For the rest of this paper we fix an *OBDA instance* $(\mathcal{T}, \mathcal{M}, \Sigma, \mathcal{D})$, where $\mathcal{T}$ is an OWL 2 QL ontology, Σ 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 Σ, and $\mathcal{D}$ is a database instance that satisfies the dependencies in Σ and the disjoint-

ness axioms in $\mathcal{T}$. We denote by $\mathrm{col}(\Sigma)$ the set of all columns in Σ. 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 $f(\vec{x})|_p$.

Table 1

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

$\mathrm{ShWellbore} \sqsubseteq \mathrm{Wellbore}$	$\mathrm{ExpWellbore} \sqsubseteq \mathrm{Wellbore}$
$\mathrm{SuspWellbore} \sqsubseteq \mathrm{Wellbore}$	$\mathrm{ExpWellbore} \sqcap \mathrm{ShWellbore} \sqsubseteq \bot$

3. Data Scaling for OBDA Benchmarks: VIG Approach

The *data scaling problem* introduced in [22] is formulated as follows:

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

The notion of *similarity* is application-based. Being our goal benchmarking, we define similarity in terms of query results for the queries at hand. In [22], the authors do not consider such 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

We overview the similarity measures used by VIG, and why they are important in the scenario of OBDA benchmarking.

Table 2

Queries for our running example.

$q_1(y)$	$\leftarrow \mathrm{Wellbore}(y), \mathrm{shWellboreForField}(x, y)$
$q_2(x, n, y)$	$\leftarrow \mathrm{Wellbore}(x), \mathrm{name}(x, n), \mathrm{complYear}(x, y)$

Schema Dependencies. The scaled instance $\mathcal{D}'$ should be a valid instance for Σ. 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*[1]. The current algorithm of VIG does not support multi-attribute foreign keys.

Column-based Duplicates and NULL Ratios. They respectively measure the ratio of duplicates and of nulls in a given column, and are common parameters for the cost estimation performed by query planners in databases. By default, VIG maintains them in $\mathcal{D}'$ to preserve the cost of joining 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. The column `state` in the table `exploration_wellbore` is fixed-domain, because it partitions the elements of `id` into a fixed number of classes[2]. 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 '`suspended`' must be generated for the column `state` in order to produce objects for the class SuspendedWellbore.

VIG generates values in columns according to a *uniform distribution*, that is, values in columns have all the same probability of being repeated. Replication of the distributions from $\mathcal{D}$ will be included the next releases 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 the function w used to build objects for the class Wellbore does not match with the function f used to build objects for Fields. Indeed, fields and

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

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

Table 3
Mappings from the NPD benchmark

$\mathrm{ShWellbore}(w(id))$	← `shallow_wellbores(id,name,year,fid)`
$\mathrm{ExpWellbore}(w(id))$	← `exploration_wellbores(id,name,year,state)`
$\mathrm{SuspWellbore}(w(id))$	← `exploration_wellbores(id,name,year,state),state='suspended'`
$\mathrm{Field}(f(fid))$	← `fields(fid,name)`
$\mathrm{complYear}(w(id), year)$	← `shallow_wellbores(id,name,year,fid)`
$\mathrm{name}(w(id), name)$	← `shallow_wellbores(id,name,year,fid)`
$\mathrm{complYear}(w(id), year)$	← `exploration_wellbores(id,name,year)`
$\mathrm{name}(w(id), name)$	← `exploration_wellbores(id,name,year)`
$\mathrm{shWellboreForField}(w(id), f(fid))$	← `shallow_wellbores(id,name,year,fid), fields(fid,fname)`

Results from the evaluation of the queries on the source part build predicates in the ontology. For example, each tuple (a, b, c, d) in a relation for `shallow_wellbores` generates an object $w(a)$ and an axiom $\mathrm{ShWellbore}(w(a))$ in the ontology. In the R2RML mappings for the original NPD benchmark the term $w(id)$ 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`.

wellbores are two different entities for which a join operation would be meaningless.

On the other hand, a standard OBDA translation of q_2 into SQL produces a union of CQs containing several joins between the tables `shallow_wellbores` and `exploration_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 on the target of two distinct basic mappings identifies sets of columns for which the join operation is semantically meaningful[3]. Generating data that guarantees the correct cost for these joins is crucial in order to deliver a realistic evaluation. In our example, the join between `shallow_wellbore` and `exploration_wellbore` over the attribute `id` is empty under $\mathcal{D}$ (in fact, ExpWellbore and ShWellbore are disjoint classes). 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

We now show how VIG realizes the measures described in the previous section. The building block of VIG is a *pseudo-random number generator*, that is a sequence of integers $(s_i)_{i\in\mathbb{N}}$ defined through a transition function $s_k := f(s_{k-1})$. The authors in [11] discuss a particular class of pseudo-random generators based on *multiplicative groups modulo a prime number*. Let n be the number of distinct values to generate. Let g be a generator for the multiplicative group modulo a prime number p, with $p > n$. Consider the sequence $S := \langle g^i \bmod p \mid i = 1, \ldots, p \text{ and } (g^i \bmod p) \leq n\rangle$. Then S is a *permutation* of values in the interval $[1, \ldots, n]$. Here we show how this generator is used in VIG to quickly produce data complying with foreign and primary key constraints.

The VIG algorithm consists of two phases: the *intervals creation phase*, and the *generation phase*. In the first phase the input data instance $\mathcal{D}$ is analyzed to create a set of intervals $\mathrm{ints}(C)$ associated to each column C in the database schema. Each interval $I \in ints(C)$ for a column C is a structure $[min, max]$ keeping track of a minimum and maximum integer index. In the generation phase, the pseudo random number generator is used to randomly choose indexes from each of these intervals, and for each chosen index $i \in I$, $I \in ints(C)$, an injective function g_C is applied to transform i 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 the source instance $\mathcal{D}$ to contain values as illustrated in Figure 1, and a scaling factor $s = 2$.

4.1. Intervals Creation Phase

[3] Therefore, for which a join could occur during the evaluation of a user query.

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

Initialization Phase. For each table T, VIG sets the number $\mathrm{size}(T, \mathcal{D}')$ of tuples to generate to $\mathrm{size}(T, \mathcal{D}) * s$. Then, VIG calculates the number of non-null distinct 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 $\mathrm{dist}(C, \mathcal{D}') := \mathrm{dist}(C, \mathcal{D}) * s$. 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

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

where A is a class, T is a table, $c_1, \ldots, c_n$ are constants, and marks the columns $b_1, \ldots, b_n$ as fixed-domain.

Example 4.1 *For the tables in Figure 1,* VIG *sets* $\mathrm{size}(\texttt{ew}, \mathcal{D}') = 5 * 2 = 10 = \mathrm{size}(\texttt{sw}, \mathcal{D}')$*, and* $\mathrm{size}(\texttt{wo}, \mathcal{D}') = 10 * 2 = 20$*. Values for statistics* $\mathrm{dist}(T.\texttt{id}, \mathcal{D}')$*, where* T *is one of* $\{\texttt{ew}, \texttt{sw}, \texttt{wo}\}$*, 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* $\mathrm{dist}(\texttt{ew.active}) = 2$.

Creation of Intervals. When C is a numerical column, VIG initializes $\mathrm{ints}(C)$ by 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 $I_C := [1, \mathrm{dist}(C, \mathcal{D}')]$. The elements in $\mathrm{ints}(C)$ will be 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* $I_{\texttt{ew.id}} = [2, 11]$*,* $I_{\texttt{ew.active}} = [1, 2]$*,* $I_{\texttt{sw.id}} = [1, 10]$*, and* $I_{\texttt{wo.id}} = [1, 20]$*. Then, the intervals are associated to the respective columns. Namely,* $\mathrm{ints}(\texttt{ew.id}) = \{I_{\texttt{ew.id}}\}$*, ...,* $\mathrm{ints}(\texttt{wo.id}) = \{I_{\texttt{wo.id}}\}$.

Primary Keys Satisfaction. Let $K = \{C_1, \ldots, C_n\}$ be the primary key of a table T. In order to ensure that values generated for each column through the pseudo-random generator will not lead to duplicate tuples in K, the least common multiple $\mathrm{lcm}(\mathrm{dist}(C_1, \mathcal{D}'), \ldots, \mathrm{dist}(C_n, \mathcal{D}'))$ of the number of distinct values to be generated in each column must be greater than the number $\mathrm{tuples}(T, \mathcal{D}')$ of tuples to generate for the table T. If this condition is not satisfied, then VIG ensures the condition by slightly increasing $\mathrm{dist}(C_i, \mathcal{D}')$ for some 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 $X_1(\vec{f_1}, \vec{x_1}), \ldots, X_m(\vec{f_m}, \vec{x_m})$ be the atoms defined by basic mappings in $\mathcal{M}$, where variables correspond to qualified column names[4]. Consider the set $\mathcal{F} = \bigcup_{i=1\ldots m} \{f(\vec{x}) \mid f(\vec{x})$ is a term in $X_i(\vec{f_i}, \vec{x_i})\}$ of all the terms occurring in such atoms. A set of columns $\mathfrak{pc}$ is a *pre-cluster* if there exists a function f and a valid position p in f such that $\mathfrak{pc} = \{f(\vec{x})|_p \mid f(\vec{x}) \in \mathcal{F}\}$.

Example 4.3 *For our running example, we have*

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

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

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

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 up 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 $\mathcal{M}$. 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 $\mathcal{C}(\mathfrak{pc}_1) \cap \mathcal{C}(\mathfrak{pc}_2) \neq \emptyset$, where $\mathcal{C}(\mathfrak{pc}) = \{D \in \mathrm{col}(\Sigma) \mid \text{there is a } C \in \mathfrak{pc} : D \overset{*}{\leftrightarrow} C\}$, where $\overset{*}{\leftrightarrow}$ is the reflexive, symmetric, and transitive closure of the single column foreign key relation between pairs of columns[5]. Given a pre-cluster $\mathfrak{pc}$, the set of columns $\{c \in \mathfrak{pc}' \mid \mathfrak{pc}' \overset{*}{\leftrightsquigarrow} \mathfrak{pc}\}$ 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* $\mathfrak{pc}_{w|_1} \not\overset{*}{\leftrightsquigarrow} \mathfrak{pc}_{f|_1}$. *In fact,*

$$(\mathcal{C}(\mathfrak{pc}_{w|_1}) = \mathfrak{pc}_{w|_1} \cup \{\texttt{wo.id}\}) \cap (\mathfrak{pc}_{f|_1} = \mathcal{C}(\mathfrak{pc}_{f|_1})) = \emptyset$$

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 the cluster, and creates new intervals accordingly. Formally, consider a columns cluster $\mathfrak{cc}$. Let $H \subseteq \mathfrak{cc}$ be a set of columns, and the set $\mathcal{K}_H := \{K \mid H \subset K \subseteq \mathfrak{cc}\}$ of strict super-sets of H. For each such H, VIG creates an interval I_H such that $|I_H| := |\bigcap_{C \in H} C^{\mathcal{D}} \setminus (\bigcup_{K \in \mathcal{K}_H} \bigcap_{C \in K} C^{\mathcal{D}})| * s$, 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* $\mathcal{K}_E = \{ES\} = \mathcal{K}_S$ *and* $\mathcal{K}_{ES} = \emptyset$.

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

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

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.*

[5] Remember that VIG does not allow for multi-attribute foreign keys.

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 Y's should be greater than the X'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 $\subseteq$):
$\forall I \in S.\, \forall C_1 \in (\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}).\, \forall C_1 \subseteq C_2.\, X_{\langle C_1,I\rangle} \geq X_{\langle C_2,I\rangle}$
$\forall I \in S.\, \forall C_1 \in (\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}).\, \forall C_2 \subseteq C_1.\, X_{\langle C_2,I\rangle} \geq X_{\langle C_1,I\rangle}$
$\forall I \in S.\, \forall C_1 \in (\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}).\, \forall C_1 \subseteq C_2.\, Y_{\langle C_1,I\rangle} \leq Y_{\langle C_2,I\rangle}$
$\forall I \in S.\, \forall C_1 \in (\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}).\, \forall C_2 \subseteq C_1.\, Y_{\langle C_2,I\rangle} \leq Y_{\langle C_1,I\rangle}$
Width of the Intervals:
$\forall C \in (\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}).\, \sum_{I \in S} Y_{\langle C,I\rangle} - X_{\langle C,I\rangle} = \lvert C\rvert$

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.

Foreign Keys Satisfaction. 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 $\mathfrak{cc}$. VIG encodes the problem of finding intervals for these columns that satisfy the number of distinct values and the foreign key constraints into a *constraint satisfaction problem (CSP)* [1] as show in Table 4. The CSP problem can be solved by any off-the-shelf constraint solver, e.g., Choco [18].

Example 4.6 *At this phase, 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 `so.id` $\subseteq$ `wo.id`, *because the index* 21 *belongs*

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:

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

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 Y's should be greater than the X's:

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

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:

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

to $I_{\{\texttt{so.id}\}}$ but not $I_{\{\texttt{wo.id}\}}$. 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, $X_{\langle C, I_{\texttt{aux}} \rangle} = Y_{\langle C, I_{\texttt{aux}} \rangle}$, 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 $X_{C,I} = Y_{C,I}$.

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 $\texttt{wo.id}$.*

4.2. Generation Phase

Generation. At this point, each column in $\mathrm{col}(\Sigma)$ 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 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 NULLS 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. With respect to our running example, such information is:*

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

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

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. 2. The scaled instance $\mathcal{D}'$.

- 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 Indeces 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 assume the function* $g : \{1, 2\} \to \{\texttt{'true'}, \texttt{'false'}\}$ *such that* $g(1) = \texttt{'true'}$ *and* $g(2) = \texttt{'false'}$.

Figure 2 contains 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.*

5. VIG in Action

The data generation techniques presented in previous sections have been implemented in the VIG system, which is available on GitHub[6] as a Java maven project, and comes with documentation in form of wiki pages. VIG was initially implement as part of the NPD benchmark [14]. Now it is a mature implementation delivered since two years, licensed under Apache 2.0, and maintained at the Free University of Bozen-Bolzano.

In this section we present an 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 multiple OBDA systems (Ontop [5] and D2RQ [2]), backed by different relational engines (MySQL and PostgreSQL). Moreover, we also compare VIG scaled data to randomly generated data.

For the evaluation, we provide three experiments. In the first experiment we use the BSBM benchmark [3] 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 [10] about authors and publications, and compare it to the data generated by VIG. The last experiment, that uses the NPD benchmark , focuses on testing the impact of the mappings analysis on the quality of the scaled data.

Experiments Setting. For our experiments we used two different OBDA systems, namely D2RQ [2] v0.8.1 and Ontop [5] v1.18.0. We used PostgreSQL v9.6.2 and MySQL v5.7.17 as underlying database engines. The hardware used is a machine with 4 Intel(R) Xeon(R) E5-2680 0 @ 2.70GHz processors and 8 GB of RAM. The OS is Ubuntu 16.04 LTS.

The experiments were ran in the OBDA-Mixer system[7], which is an automated testing platform, used in the NPD benchmark, that instantiates a set of template-queries with values from the data instance to be tested against an 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 ran 4 times: 1 warm-up run and 3 test runs. Each query was executed with a timeout of 20 minutes. We treat failed executions as timeouts.

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

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

Table 6
Overview of the BSBM Experiment (D2RQ-MySQL)

db	avg(ex_t) msec.	qmpH	avg(mix_t) msec.	[σ]
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]

All the material used for the experiments can be found online[8].

5.1. 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 a scale parameter given in terms of number of products. The queries contain placeholders that are instantiated by actual values during the test phase.

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.

Resources Consumption Comparison. Figure 3 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.

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

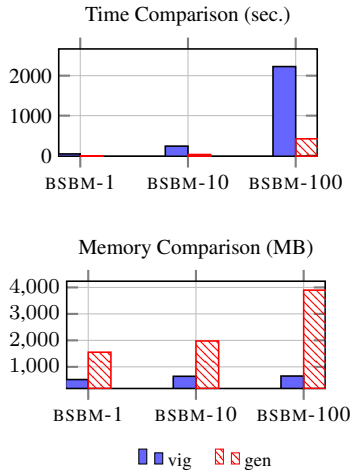

Fig. 3. Generation Time and Memory Comparison

Table 7
Overview of the BSBM Experiment (D2RQ-PostgreSQL)

db	avg(ex_t) msec.	qmpH	avg(mix_t) msec.	[σ]
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]

Benchmark Queries Comparison. We compare the execution times for the queries in the BSBM benchmark evaluated over the instances 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, so as to quantify the impact of the measures maintained by VIG on the task of approximating (w.r.t. task of benchmarking) the data produced by NTV.

The experiment was ran 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

Table 8
Overview of the BSBM Experiment (Ontop-MySQL)

db	avg(ex_t) msec.	qmpH	avg(mix_t) msec.	[σ]
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 9
Overview of the BSBM Experiment (Ontop-PostgreSQL)

db	avg(ex_t) msec.	qmpH	avg(mix_t) msec.	[σ]
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]

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}
```

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 publisher does not contribute in 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 cannot be addressed even by supporting multi-attribute foreign keys, because foreign keys must always refer to a primary key. For these reasons, we have changed the problematic URI template into an unary template by removing the redundant column `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.

Tables from 6 to 9 contain the results of the experiment in terms of various performance measures, namely the average execution time for the queries in mix ($\mathrm{avg}(ex\_t)$), the number of query mixes per hour ($qmpH$), the average mix time ($\mathrm{avg}(mix\_t)$), and the standard deviation calculated over the mix times ([σ]). We observe that the measured performance for queries executed over the instances produced by VIG 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 8 and 9), 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 6 and 7) 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

Table 10
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%

that the reason for this odd behavior is due to the fact that D2RQ is substantially (two orders of magnitude) slower than Ontop when tested against the BSBM benchmark: in D2RQ many queries timeout, 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.

Predicates Deviation Analysis. Table 10 shows the deviation, in terms of number of elements for each predicate (class, object 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 respectively report 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 which are built from tables that NTV, contrary to VIG, does not scale according to the scale factor. This aspect will be further discussed at the end of this section.

5.2. DBLP Experiment

The DBLP computer science bibliography[9] is an online 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[10]. 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.

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).

Benchmark Queries Comparison. We manually crafted a set of 16 queries, and evaluated them against the instances produced by VIG, 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[11].

Tables from 11 to 14 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. Moreover, we also observe that queries evaluated over the instance DBLP-1-RAND, obtained with the random generator, have substantially different execution times than the ones evaluated in either DBLP or DBLP-1-VIG (up to 2 orders of magnitude faster for D2RQ/MySQL, in Table 11). Interestingly, the difference is not so extreme for the tests with Ontop. In fact, Tables 13 and 14 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

[9] http://dblp.uni-trier.de/

[10] http://dblp.l3s.de/dblp++.php

[11] https://github.com/RMLio/D2RQ_to_R2RML

Table 11
Overview of the DBLP Experiment (D2RQ-MySQL)

db	avg(ex_t) msec.	qmpH	avg(mix_t) msec.	[σ]
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 12
Overview of the DBLP Experiment (D2RQ-PostgreSQL)

db	avg(ex_t) msec.	qmpH	avg(mix_t) msec.	[σ]
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 13
Overview of the DBLP Experiment (Ontop-MySQL)

db	avg(ex_t) msec.	qmpH	avg(mix_t) msec.	[σ]
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]

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 RAND already behaves significantly different from the original data and VIG at the scale factor 1, 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.

Predicates Deviation Analysis. Table 15 shows the deviation, in terms of number of elements for each predicate (class, object or data property) in the ontology, between the two instances 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. This aspect will be further discussed at the end of this section.

Table 14
Overview of the DBLP Experiment (Ontop-PostgreSQL)

db	avg(ex_t) msec.	qmpH	avg(mix_t) msec.	[σ]
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]

Table 15
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 16
Selectivity Analysis

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

5.3. NPD Experiment

The NPD Benchmark [14] is a benchmark for OBDA systems based on the the Norwegian Petroleum Directorate (NPD) FactPages [12]. The benchmark comes with an ontology, a set of mappings containing thousands of mapping assertions, and 30 SPARQL queries.

This experiment focuses on testing the impact of the mappings analysis on the quality of the scaled data.

[12] `http://factpages.npd.no/factpages`.

Table 17
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

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.

Table 16 contains the selectivities (i.e., number of results) of all four possible 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 $\mathrm{ExpWellbore}$, $\mathrm{ShWellbore}$. The class $\mathrm{DevWellbore}$ is mapped in a similar way) produce a violation of the disjointness constraints between these classes in the NPD ontology.

Table 17 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, that in turn derive from the wrong selectivities discussed in the previous paragraph.

5.4. Discussions and Limitations

In these three 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 will be similar *only* in terms of these measures. 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. 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 15 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 10, then the scaled instances might significantly diverge from the real ones.

6. 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* [13].

UpSizeR [22] replicates two kinds of distributions observed on the values for the key columns, called *joint degree distribution* and *joint distribution over co-clusters*[14]. However, this requires several assumptions to be made on the Σ, 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. A strictly related approach is *Rex* [4], which provides, through the use of dictionaries, a better handling of the content for non-key columns.

In terms of similarity measures, the approach closest to VIG is *RSGen* [21], that also considers measures like

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

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

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 [20]. Notably, this approach is able to produce *realistic* text fields by relying on machine-learning techniques based on Markov chains.

In *RDF graph scaling* [19], 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. We see this as a meaningful extension for VIG, and we are currently studying the problem of how this could be applied in an OBDA context.

Observe that all the approaches above do not consider ontologies nor 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, cannot are not taken into consideration by any of them. This leads to problems like the one we discussed through our running example, and for which we showed how it affects the benchmarking analysis in Section 5.

7. 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* is the data produced by VIG 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 instances, 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 measures conflict with the current feature of constant time for generation of tuples. In fact, many of them require access to previously generated tuples in order to be calculated (e.g., joint-degree distribution [22]).

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 it as an interesting research problem to be addressed in the future.

Acknowledgment This paper is supported by the EU project Optique FP7-318338 and UNIBZ RTD Project OBDAM.

References

[1] Apt, K.: Principles of Constraint Programming. Cambridge University Press, New York, NY, USA (2003)
[2] Bizer, C.: D2RQ - treating non-RDF databases as virtual RDF graphs. In: Proceedings of the 3rd International Semantic Web Conference (ISWC 2004) (2004)
[3] Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. Int. J. on Semantic Web and Information Systems 5(2), 1–24 (2009)
[4] Buda, T., Cerqueus, T., Murphy, J., Kristiansen, M.: ReX: Extrapolating relational data in a representative way. In: Maneth, S. (ed.) Data Science, LNCS, vol. 9147, pp. 95–107 (2015)
[5] Calvanese, D., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., Xiao, G.: Ontop: Answering SPARQL queries over relational databases. Semantic Web J. (2016), to appear
[6] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rodriguez-Muro, M., Rosati, R.: Ontologies and databases: The *DL-Lite* approach. In: Tessaris, S., Franconi, E. (eds.) RW 2009 Tutorial Lectures, LNCS, vol. 5689, pp. 255–356. Springer (2009)
[7] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rosati, R.: Linking data to ontologies: The description logic *DL-Lite*$_a$. In: Proc. of OWLED 2006. CEUR, ceur-ws.org, vol. 216 (2006)
[8] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. J. of Automated Reasoning 39(3), 385–429 (2007)
[9] Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF mapping language. W3C Recommendation, W3C (Sep 2012), available at http://www.w3.org/TR/r2rml/
[10] Diederich, J., Balke, W., Thaden, U.: Demonstrating the semantic growbag: automatically creating topic facets for faceteddblp. In: Rasmussen, E.M., Larson, R.R., Toms, E.G., Sugimoto, S. (eds.) ACM/IEEE Joint Conference on Digital Libraries, JCDL 2007, Vancouver, BC, Canada, June 18-23, 2007, Proceedings. p. 505. ACM (2007)
[11] Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly generating billion-record synthetic databases. In: Proc. of ACM SIGMOD. pp. 243–252. ACM (1994)

[12] Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C Recommendation, W3C (Mar 2013), available at http://www.w3.org/TR/sparql11-query

[13] Kontchakov, R., Rezk, M., Rodriguez-Muro, M., Xiao, G., Zakharyaschev, M.: Answering SPARQL queries over databases under OWL 2 QL entailment regime. In: Proc. of ISWC 2014. LNCS, vol. 8796, pp. 552–567. Springer (2014)

[14] Lanti, D., Rezk, M., Xiao, G., Calvanese, D.: The NPD benchmark: Reality check for OBDA systems. In: Proc. of EDBT 2015. pp. 617–628. OpenProceedings.org (2015)

[15] Lanti, D., Xiao, G., Calvanese, D.: Fast and simple data scaling for obda benchmarks. In: Proc. of Workshop on Benchmarking Linked Data (BLINK) (2016)

[16] Lanti, D., Xiao, G., Calvanese, D.: VIG. https://github.com/ontop/vig (2016)

[17] Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 Web Ontology Language profiles (second edition). W3C Recommendation, W3C (Dec 2012), available at http://www.w3.org/TR/owl2-profiles/

[18] Prud'homme, C., Fages, J.G., Lorca, X.: Choco Documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. (2015), http://www.choco-solver.org/, available at http://www.choco-solver.org/

[19] Qiao, S., Özsoyoğlu, Z.M.: RBench: Application-specific RDF benchmarking. In: Proc. of ACM SIGMOD. pp. 1825–1838 (2015)

[20] Rabl, T., Danisch, M., Frank, M., Schindler, S., Jacobsen, H.A.: Just can't get enough: Synthesizing big data. In: Proc. of ACM SIGMOD. pp. 1457–1462 (2015)

[21] Shen, E., Antova, L.: Reversing statistics for scalable test databases generation. In: Proc. of DBTest. pp. 7:1–7:6 (2013)

[22] Tay, Y., Dai, B.T., Wang, D.T., Sun, E.Y., Lin, Y., Lin, Y.: UpSizeR: Synthetically scaling an empirical relational database. Information Systems 38(8), 1168 – 1183 (2013)