Fast and Simple Data Scaling for OBDA Benchmarks

Davide Lanti, Guohui Xiao, and Diego Calvanese

Free University of Bozen-Bolzano, Italy

Abstract. In this paper we describe VIG, a data scaler for OBDA benchmarks. Data scaling is a relatively recent approach, proposed in the database community, that allows for quickly scaling an input data instance to n 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. VIG has been successfully used in the NPD benchmark, but it provides a general approach that can be re-used to scale any data instance in any OBDA setting.

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) [6] 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 [12]). 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 [12].

Data scaling [19] 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 [13]. 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.

We have evaluated VIG in the task of generating data for both the Berlin SPARQL Benchmark (BSBM) [2] and the NPD Benchmark [12]. In the first, we measured how the data produced by VIG compares to the one produced by the native ad-hoc BSBM generator. In the latter 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 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.

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 [5]. Moreover, we assume familiarity with basic notions from probability calculus and statistics.

The W3C standard ontology language in OBDA is OWL 2 QL [14]. For the sake of conciseness, we consider here its mathematical underpinning *DL-Lite*$_{\mathcal{R}}$ [7]. 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 [10], with queries evaluated under the OWL 2 QL entailment regime [11]. 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

Table 1: Portion of the ontology for the NPD benchmark. The first three axioms (left to right) state that the classes "ShallowWellbore", "ExplorationWellbore", and "SuspendedWellbore" are subclasses of the class "Wellbore". The fourth axiom states that the classes "ExplorationWellbore" and "ShallowWellbore" are disjoint.

ShallowWellbore $\sqsubseteq$ Wellbore	ExplorationWellbore $\sqsubseteq$ Wellbore
SuspendedWellbore $\sqsubseteq$ Wellbore	ExplorationWellbore $\sqcap$ ShallowWellbore $\sqsubseteq \bot$

Table 2: Queries for our running example.

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

queries containing exactly one BGP, we will use the more concise syntax for CQs rather than the SPARQL syntax. Table 2 shows 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 [8], 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 $X(\boldsymbol{f}, \boldsymbol{x}) \leftarrow \mathrm{conj}(\boldsymbol{y})$, consisting of a *target* part $X(\boldsymbol{f}, \boldsymbol{x})$, which is an atom over function symbols $\boldsymbol{f}$ (also called *templates*) and variables $\boldsymbol{x} \subseteq \boldsymbol{y}$, and a *source* part $\mathrm{conj}(\boldsymbol{y})$, which is a CQ whose output variables are $\boldsymbol{y}$. 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 disjointness 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 $f(\boldsymbol{x})$, where $\boldsymbol{x} = (x_1, \ldots, x_p, \ldots, x_n)$, we denote the argument x_p at position p by $f(\boldsymbol{x})|_p$.

3 Data Scaling for OBDA Benchmarks: VIG Approach

The *data scaling problem* introduced in [19] is formulated as follows:

Definition 1 (Data Scaling Problem). *Given a dataset $\mathcal{D}$, produce a dataset $\mathcal{D}'$ 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 query results for the queries at hand. In [19], 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.

Table 3: Mappings from the NPD benchmark. Results from the evaluation of the queries on the source part build predicates in the ontology. For example, each triple (a, b, c) in a relation for `shallow_wellbores` corresponds to a predicate $\mathrm{ShallowWellbore}(w(a))$ 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`.

Ontology predicate	Source query
$\mathrm{ShallowWellbore}(w(id))$	← `shallow_wellbores(id,name,year,fid)`
$\mathrm{ExplorationWellbore}(w(id))$	← `exploration_wellbores(id,name,year,state)`
$\mathrm{SuspendedWellbore}(w(id))$	← `exploration_wellbores(id,name,year,state),` `state='suspended'`
$\mathrm{Field}(f(fid))$	← `fields(fid,name)`
$\mathrm{completionYear}(w(id), year)$	← `shallow_wellbores(id,name,year,fid)`
$\mathrm{name}(w(id), name)$	← `shallow_wellbores(id,name,year,fid)`
$\mathrm{completionYear}(w(id), year)$	← `exploration_wellbores(id,name,year)`
$\mathrm{name}(w(id), name)$	← `exploration_wellbores(id,name,year)`
$\mathrm{shallowWellboreForField}(w(id), f(fid))$	← `shallow_wellbores(id,name,year,fid),` `fields(fid,fname)`

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.

Schema Dependencies. $\mathcal{D}'$ 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 implementation 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.

[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.

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 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 two tables `shallow_wellbores` and `exploration_wellbores`. This is possible only because the mappings for Wellbore, name, and completionYear 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 `id` is empty under $\mathcal{D}$ (because ExplorationWellbore and ShallowWellbore 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 [9] 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 $S := \langle g^i \ mod \ p \mid i = 1, \ldots, p$ and $(g^i \ mod \ p) \leq n \rangle$. 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 can be split into 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 $s = 2$.

[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`.

4.1 Intervals Creation Phase

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

$$\mathit{ClassName}(f(\boldsymbol{a})) \leftarrow \mathit{table_name}(\boldsymbol{b}), b_1 = l_1, \ldots, b_n = l_n$$

where $l_1, \ldots, l_n$ are literal values, and marks the columns $b_1, \ldots, b_n$ as fixed-domain.

Example 1. For the tables in Figure 1, VIG sets $\mathrm{size}(\mathtt{ew}, \mathcal{D}') = 5{*}2 = 10 = \mathrm{size}(\mathtt{sw}, \mathcal{D}')$, and $\mathrm{size}(\mathtt{wo}, \mathcal{D}') = 10 * 2 = 20$. Values for statistics $\mathrm{dist}(T.\mathtt{id}, \mathcal{D}')$, where T is one of $\{\mathtt{ew,\ sw,\ wo}\}$, 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 $\mathrm{ints}(C)$ by the interval $I_C := [\mathit{min}(C, \mathcal{D}), \mathit{min}(C, \mathcal{D}) + \mathrm{dist}(C, \mathcal{D}') - 1]$ of distinct values to be generated, where $\mathit{min}(C, \mathcal{D})$ 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 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, $\mathrm{ints}(\mathtt{ew.id}) = \{I_{\mathtt{ew.id}}\}$, …, $\mathrm{ints}(\mathtt{wo.id}) = \{I_{\mathtt{wo.id}}\}$.

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 $\text{dist}(C_i, \mathcal{D}')$ 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 $X_1(\boldsymbol{f_1}, \boldsymbol{x_1}), \ldots, X_m(\boldsymbol{f_m}, \boldsymbol{x_m})$ be the atoms defined by basic mappings in $\mathcal{M}$, where variables correspond to qualified column names[4]. Consider the set $\mathcal{F} = \bigcup_{i=1\ldots m} \{f(\boldsymbol{x}) \mid f(\boldsymbol{x}) \text{ is a term in } X_i(\boldsymbol{f_i}, \boldsymbol{x_i})\}$ 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 $\mathfrak{pc} = \{\mathrm{f}(\boldsymbol{x})|_{\mathrm{p}} \mid f(\boldsymbol{x}) \in \mathcal{F}\}$.

Example 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 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 $\mathcal{C}(\mathfrak{pc}) = \{D \in \text{col}(\Sigma) \mid \text{there is a } C \in \mathfrak{pc} : D \overset{*}{\leftrightarrow} C\}$, 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. 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 $\text{ints}(C)$ for all $C \in H$. Boundaries for all intervals I_H are set in a way that they do not overlap.

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

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

Table 4: CSP Program for the Choco Solver. In the following, 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.

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:
$\forall I \in S.\, \forall C \in \mathcal{C}(\mathfrak{cc}).\; I \in \text{ints}(C) \Rightarrow X_{\langle C,I\rangle} = I.min, Y_{\langle C,I\rangle} = I.max$
Set Boundaries for Known Empty Intervals:
$\forall I \in S.\, \forall C \in \mathfrak{cc}.\; I \notin \text{ints}(C) \Rightarrow X_{\langle C,I\rangle} = Y_{\langle C,I\rangle}$
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:
$\forall C \in (\mathcal{C}(\mathfrak{cc}) \setminus \mathfrak{cc}).\; \sum_{I \in S} Y_{\langle C,I\rangle} - X_{\langle C,I\rangle} = |C|$

Example 5. Consider the columns cluster $\mathfrak{pc}_{w|_1}$. There are three non-empty subsets, 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.

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], which can be solved by any off-the-shelf constraint solver. In VIG, we use Choco [15].

Example 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 $\texttt{so.id} \subseteq \texttt{wo.id}$, because the index 21 belongs to $I_{\{\texttt{so.id}\}}$ but not $I_{\{\texttt{wo.id}\}}$. Moreover, $\texttt{wo.id} \in \mathcal{C}(\mathfrak{pc}_{w|_1}) \setminus \mathfrak{pc}_{w|_1}$.

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

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

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

Therefore, VIG encodes the problem of finding the right boundaries for $I_{\texttt{wo.id}}$ into the CSP in Table 5.

Any solution for the above 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 `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

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 $\mathcal{D}'$.

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 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:

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

- *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 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 $\texttt{ew.active}$, we assume the function $g : \{1, 2\} \to \{\texttt{'true', 'false'}\}$ 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 as a Java maven project, and comes with extensive documentation in form of wiki pages. VIG, a mature implementation delivered since two years together with the NPD benchmark, is licensed under Apache 2.0, and is maintained at the Free University of Bozen-Bolzano.

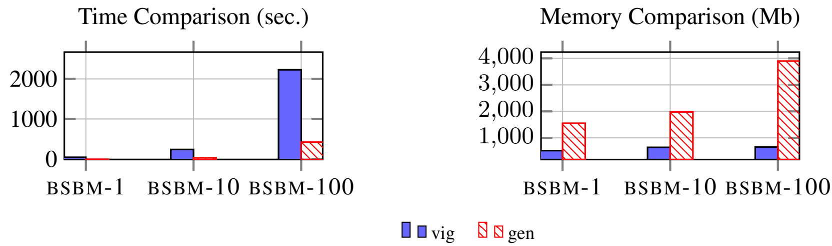

Fig. 3: Generation Time and Memory Comparison

In this section we present two evaluations of VIG. In the first evaluation we use the BSBM benchmark to compare VIG with the native ad-hoc BSBM date generator. In the second evaluation we use the NPD benchmark to show the impact, on the quality of the produced data, of the mappings analysis discussed in the previous section.

5.1 The Berlin SPARQL Benchmark

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 parametric queries, an ontology, mappings, and an ad-hoc data generator (GEN) 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 GEN), and $g \in \{$VIG,GEN$\}$ indicates the generator used to produce the instance.

The details of the experiment, as well as the material to reproduce it, are available online[6]. Queries were executed through the *Ontop* OBDA system [4], over a MySQL back-end. All the experiments have been ran on a HP Proliant server with 2 Intel Xeon X5690 CPUs (24 cores @3.47GHz), 106 GB of RAM and a 1 TB 15K RPM HD. The OS is Ubuntu 12.04 LTS.

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 GEN 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.

Benchmark Queries Comparison. In this paragraph we compare the execution times for the queries in the BSBM benchmark evaluated over the instances produced by VIG/GEN. Additionally, we here consider three additional data instances created with

[6] `https://github.com/ontop/ontop-examples/tree/master/blink-2016-vig`

Table 6: Overview of the BSBM Experiment

db	avg(ex_time) msec.	avg(out_time) msec.	avg(res_size) msec.	qmpH	avg(mix_time) [σ] msec.
BSBM-1-GEN	87	6	1425	4285	840 [101.747]
BSBM-1-VIG	77	3	841	4972	724 [85.387]
BSBM-1-RAND	40	4	887	9090	396 [94.695]
BSBM-10-GEN	628	29	11175	608	5916 [444.489]
BSBM-10-VIG	681	29	13429	563	6388 [606.057]
BSBM-10-RAND	175	19	8322	2061	1746 [551.06]
BSBM-100-GEN	6020	271	122169	63	56620 [5946.65]
BSBM-100-VIG	6022	212	83875	64	56117 [4508.37]
BSBM-100-RAND	1676	242	117259	208	17254 [3974.39]

a third generator (RAND) that only considers database integrity constraints (primary and foreign keys) as similarity measures, 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 GEN.

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 GEN. This does not allow a fair comparison between the three generators, because it is biased towards the specific values produced by GEN. Therefore, we reused the testing platform 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 binary URI templates where one of the two arguments 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-inst:dataFromRatingSite{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, 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.

Table 7: Predicates Growth Comparison

type-db-scale	avg(dev)	dev $>$ 5% (absolute)	dev $>$ 5% (relative)
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%

Table 8: 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

Finally, we slightly modified the queries by removing the `LIMIT` modifiers, so as to compare the number of retrieved results, and in a couple of cases by modifying an excessively restricting filter condition. We point out that this modification only slightly change the size of the produced SQL translation, and that the modified queries are at least as hard as the original ones.

Table 6 contains the results of the experiment in terms of various performance measures, namely the average execution/output time and average number of results for queries in a run (mix), the number of query mixes per hour, and the average mix time. Observe that the measured performance for queries executed over the instances produced by VIG is close to the measured performance for queries executed over the instances (of comparable size) produced by GEN. Moreover, it significantly differs from the performance measured over instances produced by RAND. We argue that this difference is due to the fact that RAND does not maintain any similarity measure, apart from schema constraints, on the generated data instance.

Deviation for Predicates Growth. Table 7 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 GEN. The first column reports the average deviation, and last two columns report the absolute number and relative percentage of predicates for which the deviation was greater than 5%. Apart from a few outliers (due to some elements which are built from tables that GEN, contrary to VIG, does not scale according to the scale factor) the deviation for predicates growth is inferior to 5% for the majority of classes and properties in the ontology.

5.2 The NPD Benchmark

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

Table 9: 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

compare these 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 cannot be performed, and the one (OBDA mode) discussed in this paper where the mapping is also taken into account.

Table 8 contains the selectivities (i.e., number of results) for the joins between tables `shallow_wellbore` (abbr. sw), `exploration_wellbore` (abbr. ew), and `development_wellbore` (abbr. dw), over the source NPD dataset as well as its scaled versions of factors $1, 5$ and 50. Observe that the instances created through the DB mode produce joins of non-null selectivities, and this fact together with the definitions for these classes in the mappings (see Table 3 for the mapping assertions of ExplorationWellbore and ShallowWellbore; class DevelopmentWellbore is mapped in a similar way) produce a violation of the disjointness constraints in the ontology between the classes ShallowWellbore, ExplorationWellbore, and DevelopmentWellbore.

Table 9 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.

Limitations. 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 the BSBM benchmark (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. Moreover, an intrinsic weakness 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 7, 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* [7].

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

UpSizeR [19] replicates two kinds of distributions observed on the values for the key columns, called *joint degree distribution* and *joint distribution over co-clusters*[8]. 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* [3], 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* [18], 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 [17]. Notably, this approach is able to produce *realistic* text fields by relying on machine-learning techniques based on Markov chains.

In *RDF graph scaling* [16], 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 both the BSBM and NPD benchmarks. In the first, we measured how *similar* is the data produced by VIG to the one produced by the native BSBM generator, obtaining encouraging results. In the latter, 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 data produced by adding support for multi-attribute foreign keys, joint degree and value distributions, and intra-row correlations (e.g., objects from SuspendedWellbore might not have a completionYear). Unfortunately, we expect that some of these measures conflict with the current feature of constant time for generation of tuples. Moreover, many of them require access to previously generated tuples in order to be calculated (e.g., joint-degree distribution [19]).

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

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 or the ontology. We are not aware of any work in this direction, but 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.

References

1. Apt, K.: Principles of Constraint Programming. Cambridge University Press, New York, NY, USA (2003)
2. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. Int. J. on Semantic Web and Information Systems 5(2), 1–24 (2009)
3. 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)
4. 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
5. 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)
6. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rosati, R.: Linking data to ontologies: The description logic $\textit{DL-Lite}_{\mathcal{A}}$. In: Proc. of OWLED 2006. CEUR, `ceur-ws.org`, vol. 216 (2006)
7. 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)
8. 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/`
9. 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)
10. Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C Recommendation, W3C (Mar 2013), available at `http://www.w3.org/TR/sparql11-query`
11. 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)
12. 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)
13. Lanti, D., Xiao, G., Calvanese, D.: VIG. `https://github.com/ontop/vig` (2016)
14. 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/`
15. 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/`
16. Qiao, S., Özsoyoğlu, Z.M.: RBench: Application-specific RDF benchmarking. In: Proc. of ACM SIGMOD. pp. 1825–1838 (2015)
17. 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)
18. Shen, E., Antova, L.: Reversing statistics for scalable test databases generation. In: Proc. of DBTest. pp. 7:1–7:6 (2013)
19. 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)